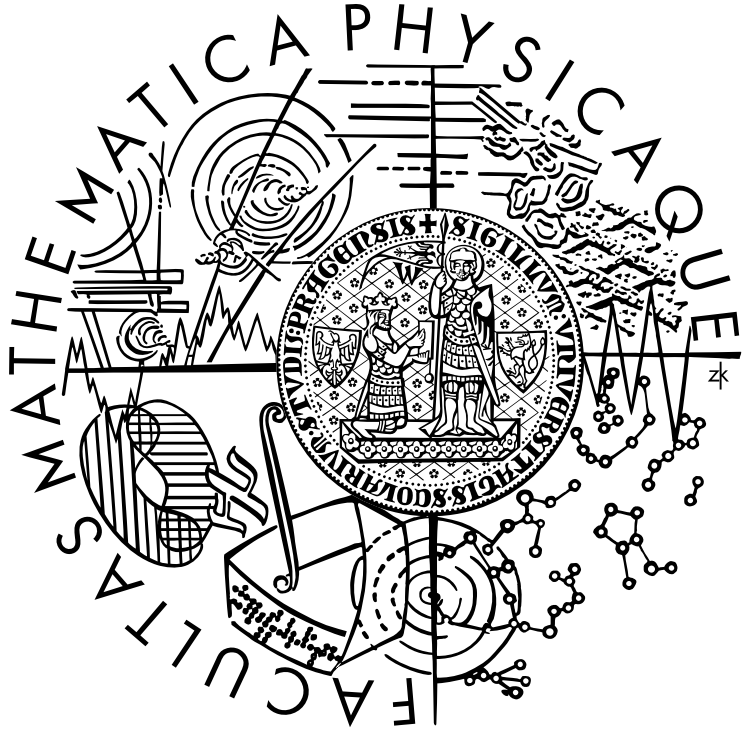
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Lenka Novotná

Automatized Data Abstraction

Department of Software Engineering
Supervisor: RNDr. Jan Kofroň
Study program: Computer Science, Software Systems

I would like to thank my advisor Jan Kofroň for his guidance and valuable suggestions, Tomáš Poch for tips on useful study materials and my family for their support throughout my studies.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

Prohlašuji, že jsem svou diplomovou práci napsala samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 16.4.2007 Lenka Novotná

Název práce: Automatized Data Abstraction
Autor: Lenka Novotná
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Jan Kofroň
E-mail vedoucího: kofron@nenya.ms.mff.cuni.cz
Abstrakt: Model checking patří mezi nejrozšířenější metody verifikace softwarových systémů. Během verifikace pomocí této metody se prochází celý stavový prostor daného systému. Stavový prostor softwarových systémů je ale obrovský a není tedy možné ho celý projít v rozumném čase. Tento problem se nazývá "state explosion problem" a jednou z metod, jak ho řešit, je použití abstrakce, kdy se konkrétní data verifikovaného systému namapují na abstraktní data. Výsledný abstraktní systém pokrývá chování původního systému nezbytné pro verifikaci, ale má výrazně menší stavový prostor, což umožňuje jeho verifikaci v rozumném čase.
Tato práce se zabývá automatickou abstrakcí dat. Shrnuje známé metody automatické abstrakce, tyto metody porovnává a na základě získaných vědomostí navrhuje novou metodu automatické abstrakce pro objektově orientované systémy.
Klíčová slova: abstrakce, model checking, predikát, abstraktní program

Title: Automatized data abstraction
Author: Lenka Novotná
Department: Department of Software Engineering
Supervisor: RNDr. Jan Kofroň
Supervisor's e-mail address: kofron@nenya.ms.mff.cuni.cz
Abstract: Model checking belongs to one of the most favourite techniques for verification of software systems. During the verification process of model checking, the whole state space of the given system is traversed. However, the state space of software systems can be huge and thus it is not possible to traverse it in reasonable amount of time. This problem is called "state explosion problem" and it can be solved using a method of abstraction that creates an abstract program from the concrete one by mapping the concrete data to abstract data. The abstract program covers all the behavior of the concrete program that is necessary for verification, but has significantly smaller state space which allows its verification in reasonable amount of time.
This work is concerned in automatized data abstraction. Three known methods for automatized data abstraction are described and compared to each other. Based on these methods a new method for automatized data abstraction of object oriented programs is designed.
Keywords: abstraction, model checking, predicate, abstract program

Contents

1 Introduction

As the progress in the field of software development is increasing, software systems are more complex and the amount of possible behaviors of these systems is huge. It is very difficult (or even impossible) to prove the correctness of such systems by hand. Testing cannot cover all the behaviors of the systems (especially when considering concurrency), and static analysis does not have to obtain accurate results. Model checking [14], [25] is a method to algorithmically verify formal systems. This method gives better results than testing and static analysis.

1.1 Model checking

The goal of model checking is to verify whether a given property is satisfied by a finite-state system. First, the model of this system has to be created and the given property is then verified over this model. When the model and the property specification are constructed, the process of verification is automatic. Each state in the state space of the model is traversed to check whether the property holds. For example, one can verify whether some variable is bigger than zero in each state. Model checking is really able to verify that a particular property is satisfied by a given model. This is possible just because model checking is based on exploration of the whole state space of the model, each state is traversed.

Model checking has been widely successful in validating and debugging designs in the hardware and protocol domains. Its usability for software systems was limited because of the amount of states in the state space of software systems models that can be huge, much larger than for models of hardware systems (software systems are typically infinite-state). The verification over such a huge state space would be very time consuming, in some cases even impossible. The size of the model increases exponentially as the number of program components grows.

The problem with a large growth of state space in the model is called *state space explosion problem*. One possible way how to solve this problem is abstraction. The real data are mapped to abstract ones to create an abstract model of the concrete software system. This abstract model has significantly smaller state space and thus it is possible to verify this model. The abstract model has to cover all behaviors of the concrete one.

There are many tools for model checking software systems, e.g. [3], [21]. These model checkers require a model (of the system to be verified) written in the programming language that is supported by them. Such programming languages are specially developed for these model checkers and thus they are not used by common programmers. To verify the program, one has to learn some of these languages first. This is the limitation that made common using of model checking impossible. To eliminate this limitation and to open up the model checking for common users, the tools were developed to allow model checking of programs written in common programming languages as the C language (e.g. [2], [4], [5]), Java (e.g. [1], [12]) etc. These tools can automatically translate the program into the language of some model checker, let the model verify by this model checker and serve the results to users in a way understandable for them.

1.2 Abstraction

Abstraction plays very important role in model checking of software systems. Models of complex software systems have usually very large state spaces and thus it would not be possible to verify them using model checking. Abstraction can significantly reduce the state space of these systems so that they can be verified.

Simply explained, to abstract a program means to map abstract variables to concrete ones and to replace concrete operations by abstract ones. The function for mapping of abstract variables to concrete ones is called an *abstraction function*. The domain of abstract variables should be much smaller than the domain of concrete ones. The smaller domain means the smaller state space of resulting abstract model.

For example, consider a variable `x` of integer (32 bit) type. If it is necessary to check only whether this variable is positive, negative or zero, it is not important to know if the variable has a value of 3 or 4. Then, this variable can be replaced by abstract variable with three-valued domain representing positive values, negative values and zero, i.e. {`pos`, `neg`, `zero`}. The domain of the original variable is significantly reduced from 2^{32} values to only 3 values for abstract variable.

There basically two approaches to data abstraction. In the first one, the abstract variables and the abstraction function have to be defined. The abstract variables are then mapped to the concrete ones using the abstraction function. The second one, called a *predicate abstraction*, tries to replace concrete variable by Boolean variable that evaluates a predicate over the concrete variable.

The example with an integer variable `x` (mentioned above) is an example of the first approach to data abstraction. The second approach (predicate abstraction) can be demonstrated on the same example. In the case of predicate abstraction, the necessity to check whether the variable `x` is positive, negative or zero is expressed by three predicates: `x > 0`, `x < 0` and `x = 0`. The concrete variable `x` will be replaced by three variables corresponding to that predicates. These variables will be evaluated by truth values of corresponding predicates.

When abstracting the program, some information can be lost due to replacement of the concrete variables by abstract ones. For example, consider an assignment `x = x + 5;` in the concrete program. If an abstract variable corresponding to `x` has the value `neg` (or, in predicate abstraction, the predicate `x < 0` holds), it is not possible to decide the value of this abstract variable after the assignment `x = x + 5;`. This value can be either `neg` or `zero` or `pos` (in predicate abstraction, the predicate `x < 0` can either hold or not). The resulting value of the abstract variable has to be chosen non-deterministically from the set of all possible values. Non-determinism increments states of the resulting model of the abstract program, because each state of the system that could come (i.e. each value that could be assigned to abstract variable) has to be considered and verified.

The abstract program is an over-approximation of the original one. It means that the abstract program covers the same behavior as the concrete one, but can cover some more behavior, that is not included in the concrete program. This is caused just by non-deterministic choices in the abstract program.

1.3 Problem statement

A choice of accurate abstraction can significantly improve the process of verification. In case of extensive software systems, abstraction can be even the only way to enable verification, because the state space of the original program (without abstraction) would be too large for model checking. On the other hand, if the abstraction is chosen in the wrong way, the verification can give lame results.

A choice of abstraction for a given concrete program is very complex challenge. An abstract program has to be as precise as possible; the amount of non-deterministic choices must be reduced to minimum. For common usage, the abstraction process needs to be maximally automated so that each user would be able to use it e.g. in scope of model checking without more than necessary cooperation.

1.4 Goal of the thesis

The goal of this thesis is to compare existing methods for automated data abstraction and, based on this comparison, to find a new method for automated abstraction. This method should be general and programming language independent. Because it is not possible to design the method absolutely independent of programming language, this method will target the object oriented programming languages. The method should also support multi-threaded programs.

1.5 Structure of the thesis

The chapter **2** describes existing tools for (semi-)automatic abstraction of programs. In chapter **2.1**, Bandera tool for semi-automatic abstraction of java programs is introduced. Chapters **2.2** and **2.3** describe two different tools for automatic predicate abstraction of programs written in the C language, the tool C2BP [22] (by Microsoft research) and the tool SATABS [18]. The chapter **3** then compares these tools from several different aspects. In chapter **4**, we try to design a new method for abstraction of object oriented programs. In chapter **5**, this method is compared to tools described in chapter **2** and a future work that can be done to improve this method is described. The thesis is concluded in chapter **6**.

2 Background

In this section, the three existing tools processing automated or semi-automated abstraction will be described. These tools are Bandera, C2BP and SATABS. Each of these tools has a little bit different approach to abstraction.

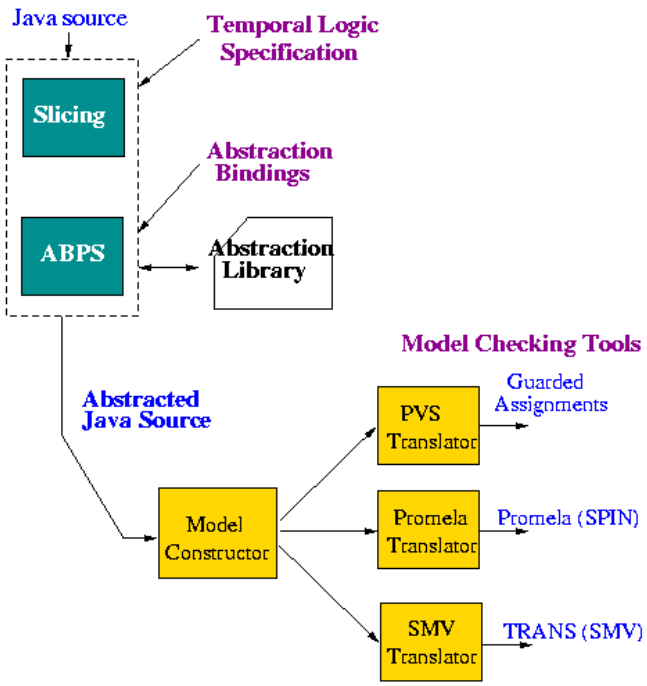
The Bandera tool focuses on Java programs. The abstraction is semi-automated and is realized by mapping variables and operations to abstract ones, which are stored in an abstraction library.

The tools C2BP and SATABS operate over the C programs. Both of these tools use the predicate abstraction. The main difference is, that C2BP uses the theorem prover for computing the abstraction, whereas the tool SATABS uses the SAT solver.

2.1 Bandera

Bandera is a tool set for model checking concurrent Java programs. It was developed by SAnToS laboratory. The goal of the Bandera project is to integrate existing techniques for processing programming language with newly developed techniques to provide automated support for the extraction of finite-state models that are safe, compact, and suitable for verification. Bandera transforms a Java program into a model so that this model can be then verified by model checkers as SPIN or SMV.

The Bandera toolset is designed as an open architecture where a variety of analysis and transformation components can be incorporated. The structure of Bandera is illustrated in Picture 1.



Picture 1: *The model of Bandera toolset. ABSP is an acronym for an Abstraction-Based Program Specialization engine (a form of partial evaluation that combines abstract interpretation and partial evaluation).*

Bandera consists of five major components:

- *property specification* – allows to define a property to check
- *program slicing* – automates the elimination of program components that are irrelevant for the property under analysis
- *program abstraction* – a combination of predicate abstraction and manual techniques for creating the definitions of abstraction is used, the definitions are stored in a library
- *verifier code generation* – transforms the sliced and abstracted program into the input format of a selected model checker
- *counter-example interpretation* – involves the mapping of low-level verifier-specific counter-examples back to Java source code

2.1.1 Abstraction overview

Abstraction in Bandera is currently implemented as half-automated, but the vision is to develop a fully automated implementation.

Bandera has four central issues:

- to provide facilities for *defining* new abstractions easily,
- to provide tool support and methodologies for *selecting* appropriate abstractions,
- *to generate* abstract programs from concrete ones, and
- *to interpret* the results of model-checking abstract programs.

The user is guided in selecting the abstraction for some data, the rest of data (usually less important ones) are abstracted automatically. Appropriate automatic abstraction is determined according to type inference based on the selections of abstraction made by user. An abstracted program is then generated by compiling abstraction definitions (classes of abstraction) into Java representations and by systematically replacing concrete operations with calls to abstract operations in the Java abstraction representation.

The procedure of verification using an abstraction if given a concrete program and temporal property could be summarized as follows:

1. Defining an abstraction mapping appropriate for the property being verified.
2. Using the abstraction mapping for transforming the temporal property into an abstract property.
3. Using the abstraction mapping for transforming the concrete program into an abstract program.
4. Verifying if the abstract program satisfies the abstract property.
5. Inferring that the concrete program satisfies the concrete property.

2.1.2 Abstract interpretation

An abstract interpretation (AI) can be informally described as a collection of three components:

- a domain of abstract values,
- an abstraction function that maps concrete program values to abstract values, and
- a collection of abstract operations (one for each concrete operation in the program).

The abstract interpretation framework establishes a rigorous semantics-based methodology for constructing abstractions so that they are *safe* in the sense that they overapproximate the set of executable behaviors of the system (it means that each executable behavior is covered by an abstract execution).

As an example of an abstract interpretation consider e.g. AI *signs*, which only keeps track on whether an integer value is positive, negative or equal to zero. The abstract domain is the powerset of the set of tokens *T={pos, neg, zero}*. The mapping function is defined as *f(n)={pos} when n>0*, *f(n)={neg} when n<0* and *f(n)={zero} when n=0*. As an example of abstract operation, let's mention the abstraction of an operation of addition:

$+_{abs}$	***Zero***	***Pos***	***Neg***
Zero	{*zero*}	{*pos*}	{*neg*}
Pos	{*pos*}	{*pos*}	{*zero, pos, neg*}
Neg	{*neg*}	{*zero, pos, neg*}	{*neg*}

The abstraction of the operation of addition was defined to have the type $+_{abs} : T \times T \to \wp(T)$, but it can be extended to the operation $\oplus_{abs} : \wp(T) \times \wp(T) \to \wp(T)$ by taking $\oplus_{abs}(S_1, S_2) \stackrel{def}{=} \cup_{t_1 \in S_1, t_2 \in S_2} +_{abs}(t_1, t_2)$.

For example $\oplus_{abs}(\{zero\}, \{pos, neg\}) = \{pos, neg\}$.

The return of multiple values (as in case of e.g. $+_{abs}(neg, pos)$) means the lack of knowledge about specific abstract values. This imprecision is interpreted in model-checker as non-deterministic choice.

The over-approximation of the AI *signs* is *safe* in the sense that each behavior of the operation + on concrete values is contained in the corresponding abstract behavior. Formally, for integers n_1 and n_2, $f(+(n_1, n_2)) \subseteq \oplus_{abs}(f(n_1), f(n_2))$.

When abstracting some variable `x` (by e.g. AI *signs*) that appears in a proposition as e.g. `x>0`, the proposition is converted to a disjunction of propositions of the form `x==a`, where *a* are the abstract values that correspond to values that imply the truth of the original proposition (e.g. `x==pos` implies `x>0`, but `x==neg` and `x==zero` do not).

2.1.3 Defining abstractions

For defining abstractions, Bandera has a special declarative language called **Bandera Abstraction Specification Language (BASL)**. This language allows defining the three components of Abstract Interpretation described in chapter **2.1.2** (a domain of abstract values, an abstraction function, and a collection of abstract operations). The abstraction specification begins with a definition of a set of tokens. The power set of this set of tokens will represent the domain of this abstraction. On the basis of the defined token set, the abstraction function mapping of concrete values to elements of the abstract domain must be defined. And finally, the BASL specification must contain a definition of an abstract operator for each corresponding concrete operator. Bandera generates the definitions of abstract operators automatically to guarantee that the definitions will be safe. The user is only allowed to create the set of tokens and the mapping function. This procedure assures that each concrete/abstract operation pair will satisfy the operation safety property and the whole Abstract Interpretation will be correct.

Example 2.1 (BASL definition of Signs AI):

```
abstraction Signs abstracts int
begin
 TOKENS = {NEG, ZERO, POS};
 abstract(n)
  begin
   n < 0   -> {NEG};
   n == 0  -> {ZERO};
   n > 0   -> {POS};
  end

 operator + add
  begin
   (NEG , NEG)   -> {NEG};
   (NEG , ZERO)  -> {NEG};
   (ZERO , NEG)  -> {NEG};
   (ZERO , ZERO) -> {ZERO};
   (ZERO , POS)  -> {POS};
   (POS , ZERO)  -> {POS};
   (POS , POS)   -> {POS};
   (_,_)         -> {NEG,ZERO,POS};
  end
end
```

Automatic generation of an abstract operator

Let *op* be a concrete binary operator, op_{abs} be its abstraction, a_1 and a_2 be a pair of abstract tokens. The process of automatic creation of an operator abstraction starts with the most general definition assuming that $op_{abs}(a_1, a_2)$ can output any of the abstract tokens (what trivially satisfies the safety property). Then, Bandera checks (using theorem prover) for each token in the output, if the safety property would still hold after eliminating the token from output. The token can be safely eliminated if the result of the concrete operation applied to concrete values can not be abstracted to that abstract token.

Example 2.2 (Automatic generation of an abstract operator):

Let's consider the derivation of $+_{abs}(neg, neg)$ in the Signs AI. At the beginning we assume $+_{abs}(neg, neg) = \{neg, zero, pos\}$ and we try to prove the following implications where *neg?(n)*, *zero?(n)*, and *pos?(n)* are the predicates from the abstraction function associated with the respective abstract token (e.g. *pos?(n)* holds iff $\alpha(n) = \{pos\}$):

- $neg?(n_1) \wedge neg?(n_2) \Rightarrow \neg neg?(+(n_1, n_2))$
- $neg?(n_1) \wedge neg?(n_2) \Rightarrow \neg zero?(+(n_1, n_2))$
- $neg?(n_1) \wedge neg?(n_2) \Rightarrow \neg pos?(+(n_1, n_2))$

The theorem prover establishes that the second and third implications are true for any integer values n_1 and n_2. From this, Bandera infers that the output of $+_{abs}(neg, neg)$ should not include *zero* or *pos*, therefore the output should be *{neg}*.

BASL includes also formats for specifying abstract interpretations of non-basic data types as classes and arrays. Specifying abstract interpretation of a class subsists in possibility for users to assign an abstract interpretation to individual fields of the class. Users are currently not allowed to specify abstract versions of class's methods, these are derived automatically.

In case of arrays, users specify an integer abstraction for the array index and an abstraction for the component type.

For example, let's consider an array `PersonInfo person[k]`. If somebody would like to verify some property of a person stored in the array at e.g. position 4, the appropriate abstraction of an array's index would be a set of tokens *{belowfour, four, abovefour}*. The abstraction of the component type `PersonInfo` would depend on fields from `PersonInfo` relevant to verified property. The resulting field would be a 3-field array `PersonInfoAbs personA[3]`, where `personA[belowfour]` would summarize all the information of fields 0 through 3 of the original array `person` and `personA[abovefour]` would summarize all the information of fields 5 through *k* of the original array `person`.

When the required BASL specification is created, the user submits it to the abstraction library manager that compiles the representations of the abstract interpretations and enters resulting representations into an abstraction database. In this database, the abstract interpretations are organized according to concrete types that they abstract. Within the scope of each concrete type, the abstract interpretations can be sub-organized into different families. As an example of possible abstract interpretations for the type of integer can be:

- a ***range*** AI that tracks concrete values between lower and upper bounds *l* and *u* and abstracts values less than *l* and greater than *u* by using token set of the form *{belowl, l, ..., u, aboveu}*.

- a ***set*** AI that tracks given concrete values and abstracts all the values other than given ones into one abstract value. A token set for abstracting all the values except 1, 2 and 3 would look like *{one, two, three, other}*.

- a ***modulo-k*** AI that merges all integers which have the same remainder when divided by *k*. A token set of e.g. *modulo-3* would be of form *{mod_zero, mod_one, mod_two}*.

- an ***even-odd*** AI that has a token set *{even, odd}* and actually corresponds to *modulo-2* AI.

- a ***point*** AI whose mapping function maps all concrete values to one abstract value *point*. The token set of this AI contains only this single value – *{point}*. The effect of this AI is to throw away all information about the data domain, what is useful when the concrete values of the data have no impact on the property being verified.

2.1.4 Process of selecting abstraction

The **optimal abstraction** could be defined as an abstraction where any finer abstraction adds irrelevant information and any coarser abstraction introduces infeasible behaviors. For example the optimal abstraction for a variable `x` that appears only in conditionals `(x==0)` and `(x>0)` would be the *signs* abstraction with token set `{neg, zero, pos}`. Using the *range(0,1)* abstraction where the `pos` value is decomposed into values `one` and `aboveone` would not be optimal, because both values `one` and `aboveone` yield true for the conditional `(x>0)` and false for the conditional `(x==0)` just as the `pos` value. Hence they provide no new information and it would be ineffective to have two values `one` and `aboveone` instead of one value `pos`. On the other hand, using the *set(0)* abstraction which collapses `neg` and `pos` values to a `nonzero` token risks introducing infeasible paths where a positive value appearing at a conditional `(x>0)` in the concrete program yields a transition to the false branch in the abstract program.

Achieving an optimal abstraction must be a compromise between a desire to compress the state space via reduction of data domain and a necessity to preserve data properties that are relevant to the property being checked.

The selection of an abstraction itself should be a process driven by relationships between data and control points mentioned in the property and data and control points mentioned in the program that can influence their execution. The consecution of exploiting these relationships can be described in following four steps:

1. **Starting with *point* AI** what means that all variables are abstracted to `point` at first.

2. **Identifying variables referenced in the property** what means that the propositions in the property to be checked can refer to variables and these variables must be then abstracted in a way that preserves the ability to decide the proposition.

3. **Selecting controlling variables** what means that there can be variables in the program that are not directly mentioned in the property to be checked, but the

variables mentioned there can be control and data dependent on these variables. Conditional expressions referencing these controlling variables suggest additional variables that should be abstracted.

4. **Selecting variables with broadest impact** what means that if there are multiple controlling variables to abstract, the ones appearing the most often in a conditional should be selected for abstraction.

In Bandera, this consecution is supported through calculation and browsing of the *program dependence graph* and *abstract type inference* described in the following sections.

2.1.5 Program dependence graph (PDG)

The program dependence graph for the given Java program is calculated by the slicing component (for more information about Bandera slicing, see [13] and [15]). Slicing is driven by the propositions in the property to be checked. This means that all definitions of all variables which appear in the propositions will be included in slicing criterion and all program variables that cannot influence the truth or falsity of the propositions will be eliminated.

The program dependence graph is created based on few types of dependences:

- data dependence,
- control dependence,
- interference dependence,
- divergence dependence,
- synchronization dependence, and
- ready dependence.

Together with dependences, a **Control Flow Graph (CFG)** is used for creating the PDG. Control Flow Graph is a flow graph *G = (N, E)* representing the program threads, where *N* is a set of statement nodes and *E* is a set of directed control-flow edges. The set *N* contains two special nodes $n_S \in N$ and $n_E \in N$, and for each $n \in N$ there is a path from start node n_S to end node n_E that includes *n*. More about CFG and slicing is in [13].

Definition (def(i), ref(i)):

- ***def(i)*** *is a set of variables* ***defined*** *at CFG node* ***i***
- ***ref(i)*** *is a set of variables* ***referenced*** *at CFG node* ***i***

Definition (data dependence):

Node n is ***data-dependent*** *on m if there is a variable v such that*

1. *there exists a non trivial path p from m to n such that for every node* $m' \in p - \{m, n\}$, $v \notin def(m')$, *and*
2. $v \in def(m) \cap ref(n)$

A node *n* is data-dependent on node *m* if, for a variable *v* referenced at *n*, a definition of (an assignment to) *v* at *m* reaches *n*. Thus, node *n* depends on node *m* because the assignment at *m* can influence a value computed at *n*.

Definition (domination, post-domination):

Node n ***dominates*** *node m in CFG G if every path from the start node* n_S *to node m passes through n.*
Node n ***post-dominates*** *node m in CFG G if every path from node m to the end node* n_E *passes through n.*

Definition (control dependence):

Node n is ***control-dependent*** *on node m if*

1. *there exists a non-trivial path p from m to n such that every node* $m' \in p - \{m, n\}$ *is post-dominated by n, and*
2. *m is not post-dominated by n.*

Control dependence information identifies the conditionals that may affect execution of a node in the slice[1]. For a node *n* to be control-dependent on *m*, *m* must have at least two immediate successors in CFG (*m* must be a conditional), and there must be two paths that connect *m* with n_E such that one contains n and the other does not.

Definition (divergence dependence):

Let a pre-divergence point be the 'decision-point' of a loop, where the condition is checked to stay in the loop or leave it. Node n is ***divergence-dependent*** *on node m if*

1. *m is a pre-divergence point,*
2. *there exists a non-trivial path p from m to n such that no node* $m' \in p - \{m, n\}$ *is a pre-divergence point.*

Divergence dependence is a variation of weak-control dependence and is used to capture the situation where an infinite loop may prevent the execution of some program node. The definition of divergence dependence allows slicing to remove infinite loops that can not infinitely delay the execution of a relevant node.

Definition (interference dependence):

A node n is ***interference-dependent*** *on node m if*

1. $\theta(n) \neq \theta(m)$, *and*
2. *there is a variable v, such that* $v \in def(m)$ *and* $v \in ref(n)$.

Interference dependence introduces dependence between node *n* and node *m* (in a different thread), if *m* defines a variable that is referenced in *n*. It captures the situation where definitions to shared variables can "reach" across threads.

---

[1] part of the program arisen from slicing

Synchronization dependence and ready dependence belong to concurrency-related dependences. These dependences are important for slicing, because if some variable is defined at node *n* inside of some critical section, then the corresponding enter-monitor and exit-monitor commands (i.e. locking) must appear in the slice. Omitting the monitor might allow shared variable interference that wasn't present in original program.

Let ***CR(n)*** be a function that maps each node *n* to the inner-most critical section in which it appears. That is, if $CR(n) = (m_1, m_2)$ then m_1 is an ***enter-monitor*** *k* command with matching ***exit-monitor*** *k* at m_2 and these two nodes form the inner-most critical region in which *n* appears.

Definition (synchronization dependence):

A node n is ***synchronization-dependent*** *on node m, if* $CR(n) = (m_1, m_2)$ *and* $m \in \{m_1, m_2\}$.

The definition says that each node in the body of a monitor is **synchronization-dependent** on the commands ***enter-monitor*** *k* and ***exit-monitor*** *k* that define the monitor.

Dependence in case of nested critical sections is captured by the transitive closure of this relation.

Let ***code(f)*** be a function mapping of a node *n* of CFG to the code for the statement that it labels.

Definition (ready dependence):

A node n is ***ready-dependent*** *on node m, if*

1. $\theta(n) = \theta(m)$ *and n is reachable from m in the CFG of* $\theta(m)$ *and code(m) =* ***enter-monitor*** *k, or*
2. $\theta(n) \neq \theta(m)$ *and code(n) =* ***enter-monitor*** *k and code(m) =* ***exit-monitor*** *k, or*
3. $\theta(n) = \theta(m)$ *and n is reachable from m and code(m) =* ***wait*** *k, or*
4. $\theta(n) \neq \theta(m)$ *and code(n) =* ***wait*** *k and* $code(m) \in \{$***notify*** $k,$ ***notify – all*** $k\}$*, or*
5. *code(n) is any statement in thread* $\theta(n)$*, and code(m) is* $\theta(n).start()$.

Ready dependence handles analogous situation as divergence dependence, but for concurrent execution. Informally, a statement *n* is ready-dependent on a statement *m* if *m*'s failure to complete (either because it is never reached or because the command ***notify*** for the command ***wait*** never occurs) can make the thread of the statement *n* (referred to as $\theta(n)$) blocked before reaching or completing *n*. The execution of *n* is thus indefinitely delayed.

The formal definition of ready dependence is overly pessimistic for a broad class of programs since it assumes that the locks will be held indefinitely. In general, locks in Java will probably not be held indefinitely. Let a ***safe lock*** be a lock that is never held indefinitely. It can be defined as a lock where all paths between matching ***enter-monitor*** *k* and ***exit-monitor*** *k* commands contain:

1. no **wait-free** indefinite loops,

2. no **wait** commands for other locks, and
3. no **enter-monitor** or **exit-monitor** commands on unsafe locks.

A **wait-free** loop is a loop which has some path through its body that does not include a **wait** command.

Bandera doesn't try to find an exact set of safe locks; instead they focus on an easily identified subset of the safe locks. Having the subset of safe locks, the definition of ready dependence can be refined. The first two conditions of this definition, that are present to capture the situation when one thread holds a lock indefinitely and blocks other process at **enter-monitor** statement on that lock, will be applied only if the lock is unsafe, since all safe locks will eventually be released.

The PDG can be visualized, navigated and queried. Querying is performed by specifying a group of program statements that will form the root of a search in PDG. Queries search upward in the PDG to designated target nodes along specified set of dependence edges. The results are presented as a set of paths in the PDG through which is possible to navigate.

Selecting controlling variables is supported by PDG query that locates all conditionals which influence the root of the search through data and control dependences.

More about slicing and PDG is in [13] and [15].

2.1.6 Abstract type inference

Bandera analyzes the abstractions selected by users to determine whether they are in conflict and to select the abstract types for the rest of variables in the program. This analysis is performed by executing a type inference algorithm.

The conflict between two abstraction selections occurs when two abstract values appear as operands in an expression and there is no meaningful way how to transfer information between those values. Let's consider e.g. an assignment `y = z;` where `y` is abstracted by *signs* AI and `z` is abstracted by *even-odd* abstraction. It is unclear how to convert the values `even` and `odd` for `z` to values `neg`, `pos` and `zero` for `y`. Another problem is when a variable with assignments to two variables of different abstraction selections appears in the program. It is then unclear which one of that two abstraction selections should be used for this variable. To solve these problems, it is possible either to introduce coercions into the program and to prioritize abstractions or to let the user change the conflicting abstraction selections.

The type inference algorithm calculates the abstract type of each program variable that is data dependent on variables for which the user made explicit abstraction selections. During this process, also the variables whose abstractions are independent on explicit selections are identified. Such variables influence the program through control or synchronization dependences rather than data dependences. There are three ways of abstraction of such variables:

- to abstract them to *point* AI to minimize the state space of the extracted model,
- to abstract them to their concrete type to maximize the precision of the extracted model, and

- to choose abstractions from the abstraction library for them.

2.1.7 Generating of an abstract program

Generation of an abstract program includes two separate steps:

- When all AIs for a program's data components are selected, the BASL specification for each of these AIs is retrieved from the abstraction library and compiled into a Java class that implements the AI's abstraction function and abstract operations.

- The concrete Java program is traversed and concrete literals and operations are replaced with calls to classes created in the previous step that implement the corresponding abstract literals and operations.

Abstract tokens are implemented as integer values and the abstract function and operations are implemented as Java class methods. Representation of set values (when it is not possible to conclude a result of an operation, more values can serve as results) is implemented using the method `Bandera.choose(bits)` that denotes a non-deterministic choice among the values. This method does not have defined concrete execution semantics. Instead, when the abstracted program is translated into the language of some concrete model checker, this method is translated into the model checker's build-in construct that means a non-deterministic choice in semantics of this model checker's language.

Example 2.3 (Java representation of BASL specification of Signs AI from example 2.1):

```
public class Signs {
 public static final int NEG  = 0;  // bit-mask 1
 public static final int ZERO = 1;  // bit-mask 2
 public static final int POS  = 2;  // bit-mask 4

 public static int abstract(int n) {
   if (n < 0) return NEG;
   if (n == 0) return ZERO;
   if (n > 0) return POS;
 }

 public static int add(int arg1, int arg2) {
   if (arg1==NEG && arg2==NEG)   return NEG;
   if (arg1==NEG && arg2==ZERO)  return NEG;
   if (arg1==ZERO && arg2==NEG)  return NEG;
   if (arg1==ZERO && arg2==ZERO) return ZERO;
   if (arg1==ZERO && arg2==POS)  return POS;
   if (arg1==POS && arg2==ZERO)  return POS;
   if (arg1==POS && arg2==POS)   return POS;
   return Bandera.choose(7);
 }
}
```

Replacing concrete variables and operations by abstract ones is relatively straightforward. The only problem is with resolving which abstract version of an operation to use when multiple AI's are selected for a program. The abstract type inference process solves this problem by attaching abstract type information to each

node in the program syntax tree in addition to propagating abstract type information to each of the program variables.

For example, the code fragment `(x + y) + 2` where the variable `x` was chosen to have an abstract type *Signs* and `y` was not abstracted, will be transformed into the code `Signs.add(Signs.add(x, Signs.abstract(y)), Signs.POS);`.

One can see that the selection of *Signs* for `x` in the innermost operation + forced the abstract version of this operation to be `Signs.add`. Since the variable `y` was not abstracted, it holds a concrete value and thus coercion `(Signs.abstract)` converts `y`'s concrete value to a *Signs* abstract value. It is similar with the outermost operation +. Since the left operand of this operation is of *Signs* abstraction type, the constant `2` in the right argument is coerced to a *Signs* abstract constant as well as the operation + is abstracted to `Signs.add`.

The abstract type inference has also some additional features as allowing users to define a lattice of abstract types ordered by user-defined coercions. This feature is useful for resolving conflicts among abstract types in operations. The lattice consists of abstract types ordered by the rule telling that an abstract type T_1 lies below another type T_2 if a coercion is defined from T_1 to T_2. Such lattices have a bottom type, e.g. the lattices for integers usually have a bottom type of concrete integers and the *point* abstraction. The lattice structure allows to define a notion of *the best (most precise) abstract typing* what is the typing returned by the inference algorithm.

2.1.8 Results of the abstraction

The abstraction computed by Bandera is an over-approximation of the original program. It means that if a specification is true for the abstracted program, it will also be true for the concrete program. However, if the specification is false for the abstracted program, it does not mean that it is false for the original program as well; the counterexample may be the result of some behavior in the abstracted program which is not present in the original program.

The interpretation of model checking's outcomes in Bandera results from the following theorem from [9]: "*Every path in the abstracted program where all assignments are deterministic is a path in the concrete program.*"

Java Path Finder model checker was enhanced with an option to look only at paths that do not refer to instructions introducing non-determinism. When the next instruction introduces non-determinism, the search algorithm backtracks. The theorem ensures that paths without non-determinism indeed correspond to paths in the concrete program. So if a counterexample is reported then it represents a real counterexample, which can be matched by a computation in the concrete program.

The problem of such approach is that the paths with non-determinism can also refer to feasible paths in the concrete program. And if some of these paths would be counterexamples, they would not be reported. So, if no counterexample is reported by model checker enhanced with this option, we cannot be sure, that no counterexamples exist in the concrete program. This option is useful in the process of model checking, but only like a supporting tool, it cannot be used as the only tool.

The whole process of results obtaining and interpreting could be described as follows:

The concrete program and the specification are abstracted using abstractions from the abstraction library, and verified using a model checker. If the result of model checking is true, then the specification is true for the concrete program. If the result is false, the generated counterexample has to be analyzed to find out if it does not correspond to some spurious behavior introduced by abstractions. Instead of trying to match the error trace to a concrete computation, the model checker is re-run to search only non-determinism free paths in the model. If the model checker finds an error trace then a guaranteed feasible counterexample is reported. Otherwise, the abstractions are too coarse and the counterexample from the first model check is used to guide the re-selection of abstractions.

2.2 C2BP

The tool C2BP is a part of the SLAM toolkit. This toolkit statically checks temporal safety properties of programs using a combination of predicate abstraction, model checking, symbolic reasoning and iterative refinement.

2.2.1 Abstraction overview

The tool C2BP implements the automatic construction algorithm for predicate abstraction of programs written in C language and similar languages. Given a program *P* and a set of predicates *E*, the tool automatically creates a Boolean program *BP(P,E)*, which represents the abstraction of original program *P*. The Boolean program *BP(P,E)* is a program written in the C language where the only type is the Boolean type and the only variables are variables representing predicates from the set of predicates *E*. The resulting Boolean program is then processed by the tool BEBOP [23], which performs interprocedural dataflow analysis using binary decision diagrams.

The tool C2BP guarantees that any feasible execution path of the original program *P* is a feasible execution path of the Boolean program. But there may be feasible execution paths of the Boolean program that are infeasible in the C program. Such paths are called spurious counterexamples and can lead to imprecision in subsequent model checking.

It's not always possible to determine the influence of the statement in program *P* on the predicate in terms of the input predicate set *E*. Such non-determinism is in Boolean program solved by non-deterministical control expression "*" which means "unknown" and represents non-deterministic selection from values TRUE and FALSE.

Computation of abstraction function for each expression in program *P* with respect to set of predicates *E* needs a theorem prover. In the worst case we have $O(2^{|E|})$ calls to the theorem prover during program abstraction. There are a lot of optimizations, some of them prefer abstraction correctness (equivalence), and other ones prefer computation speed.

2.2.2 Program preparation

It is given a program *P* in the C language, a set of predicates *E* and constants of the C language. Predicates are boolean expressions over the variables from the program *P*: $E = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$.

C2BP expects the program *P* converted to the program where:

- all intraprocedural control-flows are accomplished with if–then–else and goto statements
- no expressions contain side-effects, short-circuit evaluations and multiple pointer dereferences (single pointer dereference is allowed)
- function calls are located only at the very beginning of expressions (e.g. the expression `z = x + f(y);` would be replaced by expressions `t = f(y); z = x + t;`)

C2BP can work with all syntactic constructs of C language as pointers, structures and procedures. The main limitation is that C2BP uses a logical memory model during program analysis. The expression `p+i`, where `p` is a pointer and `i` is an integer, is modeled by C2BP as yielding a pointer value that points to the object pointed to by `p`.

C2BP automatically creates an abstraction of the program *P* with respect to the set *E*. This abstraction is a Boolean program *BP(P, E)*. The Boolean program *BP(P, E)* has identical control structure to the program *P*, but contains only boolean variables $V = \{b_1, b_2, ..., b_n\}$, where each b_i represents the predicate φ_i $(1 \leq i \leq n)$. The set of execution traces of program *BP(P, E)* is a superset of the set of execution traces of program *P*.

2.2.3 The Weakest liberal precondition

Crucial to the predicate abstraction process is a computation of the weakest preconditions.

Definition (The Weakest Liberal Precondition):

For a statement s and a predicate φ, let ***WP(s, φ)*** *denote the weakest liberal precondition of φ with respect to statement s.* ***WP(s, φ)*** *is defined as the weakest predicate whose truths before s entails the truth of φ after s terminates (if it terminates).*

> ***Example 2.4 (The Weakest Liberal Precondition – part 1):***
>
> Let "`x = e;`" be an assignment, where `x` is a scalar variable and `e` is an expression of the appropriate type. Let φ be a predicate. By definition $WP($`x = e`$, \varphi)$ is φ with all occurrences of `x` replaced with `e`, denoted φ[`e`/`x`]. E.g. $WP($`x = x + 1, x < 5`$) =$ `(x + 1) < 5` $=$ `(x < 4)`.

Suppose that the statement *s* occurs in the program *P* between program points *p* and *p'*. Let *b* be a variable corresponding to a predicate φ from the set of predicates *E* and let *b'* be a variable corresponding to the weakest liberal precondition *WP(s, φ)*. Then it is save to assign *b* the value true in the boolean program *BP(P,E)* between program points *p* and *p'* in case the boolean variable *b'* is true at program point *p*.

The problem is that if $WP(s, \varphi) \notin E$ then *b'* does not exist.

Example 2.5 (The Weakest Liberal Precondition – part 2):

E = {(x < 5), (x = 2)} and WP(x = x + 1, x < 5) = (x < 4), but (x < 4) $\notin E$.

In this case the C2BP uses the decision procedure (e.g. a theorem prover) for straightening the weakest liberal precondition to an expression over the set of predicates E. In the example above, it's possible to show that (x = 2) □ (x < 4). Therefore if (x = 2) is true before the statement "x = x + 1", then (x < 4) is true afterwards.

Formalization of predicate straightening:

A ***cube*** *over V is a conjunction* $c_{i_1} \wedge \ldots \wedge c_{i_k}$, *where each* $c_{i_j} \in \{b_{i_j}, \neg b_{i_j}\}$ *for some* $b_{i_j} \in V$. *For a variable* $b_i \in V$, *let* $\varepsilon(b_i)$ *denote the corresponding predicate* φ_i *and let* $\varepsilon(\neg b_i)$ *denote the predicate* $\neg\varphi_i$. *Extend* ε *to cubes and disjunctions of cubes in the natural way. For any predicate* φ *and set of boolean variables V, let* $\Im_V(\varphi)$ *denote the largest disjunction of cubes c over V such that* $\varepsilon(c)$ *implies* φ. *The predicate* $\varepsilon(\Im_V(\varphi))$ *represents the weakest predicate over* $\varepsilon(V)$ *that implies* φ.

Formalization of predicate weakening:

Corresponding predicate weakening $\mathcal{G}_V(\varphi)$ *is defined as* $\neg\Im_V(\neg\varphi)$. *The predicate* $\varepsilon(\mathcal{G}_V(\varphi))$ *represents the strongest predicate over* $\varepsilon(V)$ *that is implied by* φ.

Example 2.6 (The Weakest Liberal Precondition – part 3):

$\varepsilon(\Im_V(x < 4)) = (x = 2)$.

2.2.4 Predicate abstraction of assignments

Let's consider the assignment "x = e;" at label l in the program P. The program *BP(P,E)* contains parallel expression at label l. It's necessary to find out how will this assignment involve the boolean variables in *BP(P,E)*. If the expression $\Im_V(WP(x = e, \varphi_i))$ is true before label l, then the variable b_i will be **true** after the label l. If the expression $\Im_V(WP(x = e, \neg\varphi_i))$ is true before label l, then the variable b_i will be **false** after the label l. The expressions $\Im_V(WP(x = e, \varphi_i))$ and $\Im_V(WP(x = e, \neg\varphi_i))$ can't be true at the same time.

If none of these expressions holds before the label l, then the value of the variable b_i must be set non-deterministically. This situation can happen because the predicates in the set E are not strong enough to provide appropriate information, or because the theorem prover is incomplete.

The abstraction of the assignment "`x = e;`" in program *P* is the following assignment placed at label *l* in the boolean program *BP(P,E)*:

$$b_1 \ldots b_n = \texttt{choose}\ (\Im_V(WP(x = e, \varphi_1)), \Im_V(WP(x = e, \neg\varphi_1))), \ldots,$$
$$\texttt{choose}\ (\Im_V(WP(x = e, \varphi_n)), \Im_V(WP(x = e, \neg\varphi_n)))$$

where the `choose` function is always part of *BP(P,E)* and is defined as follows:

```
bool choose(bool pos, bool neg) {
      if (pos) { return true; }
      if (neg) { return false; }
      return unknown();
}
```

2.2.5 Predicate abstraction of `goto` expressions and conditions

The expression `goto` is simply copied into the boolean program *BP(P,E)*.

The condition `if-then-else` is more complicated. Consider some conditional `if(φ){...}else{...}` in program *P*. We know that at the beginning of the `then` branch in program *P* the predicate φ has to hold. Therefore, at the beginning of the corresponding `then` branch in program *BP(P,E)*, the condition $\mathcal{G}_V(\varphi)$ holds. Similarly, at the beginning of the `else` branch in program *P* the predicate $\neg\varphi$ has to hold. Therefore, the condition $\mathcal{G}_V(\neg\varphi)$ holds at the corresponding program point in *BP(P,E)*.

The abstraction of `if-then-else` condition looks generally as follows:

$$\texttt{if (*) \{ assume } (\mathcal{G}_V(\varphi))\ldots\texttt{\} else \{ assume } (\mathcal{G}_V(\neg\varphi))\ldots\texttt{\}}$$

The conditional is `*`, so both branches (`then` as well as `else`) are possible to go through. The expression `assume` is dual of the expression `assert`, `assume(φ)` never fails and the executions on which φ doesn't hold at the point of the `assume` are simply ignored.

2.2.6 Predicate abstraction of procedure calls

Let G_P be a set of all global variables of the program *P*. Each predicate from the set of predicates *E* is either global in *BP(P,E)* or local in some procedure in *BP(P,E)*. Global predicates can represent only variables from G_P.

Let E_G be a set of all global predicates from the set *E* and let V_G be the set of corresponding global boolean variables in *BP(P,E)*.

For the procedure *Q*, let E_Q be the subset of predicates from the set *E* that are local in *Q* and let V_Q be the set of corresponding local boolean variables of the procedure *Q* in *BP(P,E)*.

Let F_Q be the set of formal parameters of the procedure *Q* and let L_Q be the set of local variables of procedure *Q*.

Let $r \in L_Q \cup F_Q$ be the return value of the procedure Q (we can assume that there is only one return value, without loss of generality).

Let *vars(e)* be the set of variables referenced in expression *e* and let *drfs(e)* be the set of variables dereferenced in expression *e*.

The abstraction exploits modularity; each procedure can be abstracted given only the signatures of procedures it calls. The signature of the procedure Q can be determined in isolation of the rest of program, using E_Q.

C2BP works in 2 passes. During the first pass it determines the signature of each procedure and during the second pass it abstracts procedure calls using signatures determined in the first pass.

Definition (Signature of the procedure):

Let Q be a procedure in program P and let Q' be an abstraction of Q in BP(P,E). ***Signature of Q*** *is a quaternion (F_Q, r, E_f, E_r), where*

- F_Q *is a set of formal parameters of the procedure Q,*
- *r is a return value of the procedure Q,*
- E_f *is a set of formal parameter predicates of the procedure Q' defined as* $\{e \in E_Q \mid vars(e) \cap L_Q = \emptyset\}$ *and*
- E_r *is a set of return predicates of the procedure Q' defined as* $\{e \in E_Q \mid (r \in vars(e) \wedge (vars(e) \setminus \{r\} \cap L_Q = \emptyset)) \vee (e \in E_f \wedge (vars(e) \cap G_P \neq \emptyset \vee drfs(e) \cap F_Q \neq \emptyset))\}$.

Informally, E_f is a subset of predicates from E_Q, which do not refer to any local variable in Q. All predicates in the set $(E_Q - E_f)$ are local in Q'. E_r is a set of return predicates of the procedure Q' (in boolean procedures it is possible to have multiple return value). These return predicates have two purposes:

- to provide the caller with information about return value *r* and
- to provide the caller with information about any global variable and about a call-by-reference parameter so that local predicates of the caller can be updated correctly.

Because of the first property, E_r contains only the predicates from E_Q that mention *r* and do not mention any local variables of procedure Q (caller will not know about these local variables). And because of the second property, E_r contains the predicates from E_f that reference a global variable or dereference a formal parameter of Q.

Handling procedure calls

Consider a call `v = Q(a1...aj)` to the procedure Q that appears at label *l* of some procedure *S* in the program *P*. The abstraction *BP(P,E)* will contain a corresponding call to the procedure *Q'* at label *l*.

Let the signature of the procedure Q be the quaternion (F_Q, r, E_f, E_r). C2BP has to compute an actual value for each formal parameter predicate $e \in E_f$ to pass it into the call.

The expression e' defined as $e' = e[a_1 / f_1, a_2 / f_2 \ldots a_j / f_j]$, where $F_Q = \{f_1, \ldots, f_j\}$, represents the predicate translated into the context of calling. C2BP computes the actual value for formal parameter predicate e by the function:
$\texttt{choose}(\Im_{V_S \cup V_G}(e'), \Im_{V_S \cup V_G}(\neg e'))$.

Let's assume $E_r = \{e_1, \ldots, e_p\}$. C2BP creates p new local variables $T = \{t_1, \ldots, t_p\}$ in the procedure S' and assigns to them the return values of Q' - $t_1, \ldots, t_p = Q'(\ldots);$.

As final step, each local predicate of S, whose value may have changed by the procedure call, is updated. In addition, update must be realized at any predicate in E_S that mentions a global variable, at (possibly transitive) dereference of an actual parameter to the call, or at an alias of either of these kinds of locations.

2.2.7 Formal properties

Two formal properties relate the program P and its abstraction, the boolean program *BP(P,E)*. These properties are:

- **Soundness** – The Boolean program *BP(P,E)* is an abstraction of the program P. Each feasible path in the program P is feasible in the boolean program *BP(P,E)* as well. If Ω is a state of the program P after executing the path p, then there exists an execution of the path p in the Boolean program *BP(P,E)* ending in state Γ such that for each $1 \le i \le n$ φ_i holds in Ω iff b_i holds in Γ.

- **Precision** – Because the Boolean program that allows all paths to be feasible fulfils soundness as well, it is necessary to define another property – precision. For simple procedures without pointers the abstraction computed by C2BP is equivalent to composition of Boolean and Cartesian abstractions. Precision is improved using disjunctive completion and focus operations, both implemented in BEBOP using BDDs.

2.3 SATABS (predicate abstraction of ANSI-C using SAT solver)

The process of abstraction is similar to one used in C2BP algorithm. The main deference is that SATABS uses a SAT solver instead of theorem prover.

A SAT (Boolean satisfiability problem) instance is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The SAT instance is expressed in the conjunctive normal form (CNF). In this form, each OR term is called a clause and acts as a constraint on the possible values of its variables. The SAT solver tries to find all possible evaluations (values TRUE and FALSE) of variables in the SAT instance such that the entire expression is true.

The advantage of the SAT solver is in eliminating potentially exponential number of theorem prover calls. These are replaced by an enumeration on a single SAT instance.

SATABS encodes integer operators using bit-vector operators, so the potential arithmetic overflow is taken into account. Pointer manipulation construct, including pointer arithmetic, can also be supported. The only limitation is that recursion and dynamic memory allocation are not allowed.

2.3.1 Creating a Boolean formula for the concrete transition relation

The program is first partitioned into basic blocks composed of assignments and control-flow statements (i.e. `if`, `while`, `goto` and so on). Then a different approach has to be applied for assignments and for control-flow statements. To capture the semantics of assignments, bit-vector equations are used. These equations are not necessary for control-flow statements, because the control-flow statements do not change variable values.

Let B be a basic block containing n statements $s_1,\ldots,s_n$. Because the code was manipulated to remove function calls and empty (skip) statements, we can assume that each s_i is an assignment. Let $lhs(s_i)$ be the symbol for the left-hand side and $rhs(s_i)$ the symbol for right-hand side of the assignment s_i. Given an expression *e*, let *Vars(e)* denote the set of variables referenced in this expression. For assignments, let $Vars(s_i) = Vars(lhs(s_i)) \cup Vars(rhs(s_i))$.

First, the basic block *B* has to be transformed into a single assignment form, where each variable is assigned to only once. For this purpose, auxiliary variables for recording intermediate values has to be added. Let *v* be a variable such that $v \in Vars(s_i)$ for an assignment s_i. Let $\alpha(v,s_i)$ denote the number of assignments made to variable *v* within the basic block *B* prior to the assignment s_i:

$$\alpha(v,s_1) = 0$$
$$\forall i \geq 2 : \alpha(v,s_i) = \begin{cases} \alpha(v,s_{i-1}) + 1 : & s_{i-1} \texttt{ assigns to } v \\ \alpha(v,s_{i-1}) : & \texttt{otherwise} \end{cases}$$

Definition (renaming ρ):

Let s_i be an assignment that assigns to the variable v. Then the leftmost occurrence of v in $lhs(s_i)$ is renamed to $v_{\alpha(v,s_i)+1}$. All other occurrences of v are renamed $v_{\alpha(v,s_i)}$. Any other variable $u \in Vars(s_i)$ such that $u \neq v$ is renamed $u_{\alpha(u,s_i)}$.

Let e denote any expression (whether a part of an assignment, a whole assignment, a condition, etc.). Then $\rho(e)$ denotes the expression after this renaming.

Example 2.7:

```
x = z * x;                    x1 = z0 * x0;
y = x + 1;        ρ ⟶         y1 = x1 + 1;
x = x + y;                    x2 = x1 + y1;
```

Definition (choice operator "?"):

$$c\,?\,a : b \stackrel{\Delta}{=} \begin{cases} a: & c \neq 0 \\ b: & \texttt{otherwise} \end{cases}$$

Definition (`with` operator for arrays):

Let g be an expression of array type, i be an integer expression, and e be an expression with the type of elements in g. The operator `with` *takes g, i and e and produces an array that is identical to g, except for the content of g[i] being replaced by e. Formally, let* g' *be "*g `with` $[i] := e$*" then:*

$$g'[j] \stackrel{\Delta}{=} \begin{cases} e: & j = i \\ g[j]: & \texttt{otherwise} \end{cases}$$

Definition (`with` operator for structures):

Let s be a variable of structure type, f be a field name of this structure, and e be an expression matching the type of the filed f. The operator `with` *takes s, f and e and produces a structure that is identical to s, except for the content of s.f being replaced by e. Formally, let* s' *be "*s `with` $.f := e$*" and j be a filed name of s, then:*

$$s'.j \stackrel{\Delta}{=} \begin{cases} e: & j = f \\ s.j: & \texttt{otherwise} \end{cases}$$

An auxiliary function $\ell(l, r)$ is used for translation of an assignment into a constraint. This function maps the expression *l* for the left hand side and *r* for the right hand side into a constraint. The function is defined recursively on the structure of the expression *l*:

- If *l* is a symbol *v*, then $\ell(l, r)$ is the equality of the left hand side *l* and the right hand side *r*: $\ell(v, r) := (v = r)$
- If *l* is an array index expression *g*[*i*], then $\ell(l, r)$ is applied recursively to *g* and a new right hand side which is *g* with element *i* changed to *r*: $\ell(g[i], r) := \ell(g, g \texttt{ with } [i] := r)$
- If *l* is a structure member expression *s.f*, then $\ell(l, r)$ is applied recursively to *s* and a new right hand side which is *s* with element *f* changed to *r*: $\ell(s.f, r) := \ell(s, s \texttt{ with } .f := r)$

Definition (equation $\sigma(s_i)$):

$$\sigma(s_i) := \ell(lhs(s_i), rhs(s_i))$$

Final bit-vector equation is the conjunction of the equations $\sigma(s_i)$, formally $\bigwedge_{i=1,\ldots,n} \sigma(s_i)$.

Let v denote the version of the variable v with index 0 and v' denote the version of the variable v with the largest index, formally:

$$v := v_0$$
$$v' := v_{\alpha(v,s_{n+1})}$$

Then we can define a relation $\mathrm{T}(\bar{v},\bar{v}')$, where $\bar{v}$ is the vector of all variables v, and $\bar{v}'$ is the vector of all variables v'. This relation represents the concrete transition relation of the basic block B (expressed as a CNF formula), the vector $\bar{v}$ represents the state before the execution of the block B and the vector $\bar{v}'$ represents the state after the execution of the block B. Every solution to this equation system represents a possible computation of the basic block.

Creating the bit-vector equation for expressions with pointers will not be described here, it is closely described in [6]

2.3.2 Using SAT solver for computing the abstraction

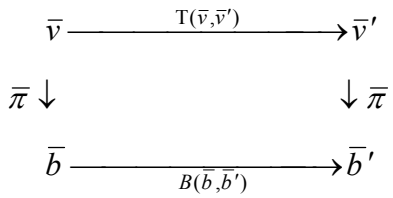
Let P denote the set of predicates over the concrete program variables. Each predicate $\pi_i \in P$ is associated with a Boolean variable b_i representing its true value. The resulting Boolean program consists of these Boolean variables. Let $\bar{\pi}$ denote the vector of predicates π_i, and $\bar{b}$ denote the vector of Boolean variables b_i. The predicates map the concrete state $\bar{v}$ into an abstract state $\bar{b}$, the function $\bar{\pi}(\bar{v})$ can be thus called the abstraction function.

Given $\mathrm{T}(\bar{v},\bar{v}')$ and P, we can create an abstract transition relation $B(\bar{b},\bar{b}')$ which is an existential abstraction of a basic block of the concrete program and relates a current state $\bar{b}$ (before the execution of the basic block B) to a next state $\bar{b}'$ (after the execution of the basic block B). Formal definition of $B(\bar{b},\bar{b}')$ using the abstraction function $\bar{\pi}$ is as follows:

$$\Gamma(\bar{b},\bar{b}',\bar{v},\bar{v}') \overset{\Delta}{=} (\bar{\pi}(\bar{v}) = \bar{b}) \wedge \mathrm{T}(\bar{v},\bar{v}') \wedge (\bar{\pi}(\bar{v}') = \bar{b}')$$
$$B(\bar{b},\bar{b}') \Leftrightarrow \exists \bar{v},\bar{v}' : \Gamma(\bar{b},\bar{b}',\bar{v},\bar{v}')$$

The abstract transition $B(\bar{b},\bar{b}')$ is actually the same as the concrete transition $\mathrm{T}(\bar{v},\bar{v}')$, but over the abstract program. It maps an abstract state $\bar{b}$ (corresponding to the state $\bar{v}$ in the concrete program) into an abstract next state $\bar{b}'$ (corresponding to the state $\bar{v}'$ in the concrete program).

Graphically, it is more clear:



Every satisfying assignment to $\Gamma(\bar{b},\bar{b}',\bar{v},\bar{v}')$ represents a concrete transition and its corresponding abstract transition. The goal is to obtain all possible satisfying assignments to the abstract variables $\bar{b}$ and $\bar{b}'$, i.e. the set $\{(\bar{b},\bar{b}') \mid B(\bar{b},\bar{b}')\}$. It is obtained by modifying the SAT solver Chaff. Detailed description of this modification is in [6].

Example 2.8 (equation system and its evaluation):

Let the basic block look like follows:

```
d = e;
e++;
```

where `d` and `e` are integer variables and let the predicates be $\pi_1 = d \,\&\, 1$ and $\pi_2 = e \,\&\, 1$. The binary operator & is the bit-wise conjunction operator, π_1 holds iff `d` is odd and π_2 holds iff `e` is odd.

The basic block is translated into the following equation system:
$d_1 = e_0 \wedge e_1 = e_0 + 1$

By adding the constraint according to the equation
$B(\bar{b},\bar{b}') \Leftrightarrow \exists \bar{v},\bar{v}' : \Gamma(\bar{b},\bar{b}',\bar{v},\bar{v}')$
we get:
$(b_1' = d_1 \,\&\, 1) \wedge (b_2' = e_1 \,\&\, 1) \wedge (b_1 = d_0 \,\&\, 1) \wedge$
$\wedge\ (b_2 = e_0 \,\&\, 1) \wedge (d_1 = e_0) \wedge (e_1 = e_0 + 1)$

The satisfying assignments for this equation system over the variables b_1, b_1', b_2, and b_2', are:

b_1	b_2	b_1'	b_2'
0	0	0	1
0	1	1	0
1	0	0	1
1	1	1	0

The Boolean program will never make a transition into a state where both `d` and `e` are odd, because `e` is equal to `d + 1` at the end of the basic block. This situation would be unavoidable in case that the next state function would be

computed separately for each Boolean variable b_i, as done by many existing tools.

2.3.3 Abstract transition relation for control-flow statements

The control-flow statements take a condition as an argument and affect only the control-flow (the program counter). They do not change the values of concrete variables, so it is not necessary to create an equation system to represent them.

Assume abstracting a specific program counter location *l* that evaluates a condition *c* and moves the program counter to location l_T if *c* holds and l_F otherwise. The goal is to generate two sets of abstract transitions, one assigning l_T to the program counter and the other assigning l_F to the program counter. All these transitions will let the abstract variables $\bar{b}$ unchanged.

First, the syntactic structure of the condition *c* must be traversed to find out, whether it contains some sub-expressions that are also the predicates in the set of predicates *P*. All such sub-expressions π_i are replaced by corresponding Boolean variables b_i. If so modified condition *c* (lets denote it c_M) references only to Boolean variables, it can be used in the abstract program. An abstract statement that assigns the program counter with l_T if $c_M(\bar{b})$ holds and l_F otherwise is generated.

If c_M still refers to some concrete variables $\bar{v}$, the SAT enumeration engine has to be used to produce the set of abstract transitions corresponding to the evaluation of *c*. The condition $c(\bar{v})$ holds in an abstract state $\bar{b}$ if and only if there is a concrete state $\bar{v}$ such that the condition holds in $\bar{v}$ and $\bar{v}$ is mapped to $\bar{b}$.

First, we will define the set pos_c of abstract states from which there is a transition that assigns the program counter with l_T:

$$pos_c = \{\bar{b} \mid \exists \bar{v} : c(\bar{v}) \wedge \bar{\pi}(\bar{v}) = \bar{b}\}$$

A dual set to the pos_c is a set neg_c of abstract states from which there is a transition that assigns the program counter with l_F. This set is not the negation of pos_c, because a single abstract state can correspond to both concrete states that satisfy *c* and concrete states that do not. The set neg_c is defined as follows:

$$neg_c = \{\bar{b} \mid \exists \bar{v} : \neg c(\bar{v}) \wedge \bar{\pi}(\bar{v}) = \bar{b}\}$$

Both of these sets - neg_c and pos_c - are computed by the SAT enumeration engine. In practice, the SAT enumeration engine is rarely required to compute these sets, because the conditions are often chosen as the Boolean predicates in the set *P*, so the modified condition c_M contains only the Boolean variables.

2.3.4 Optimization - minimizing the number of quantified variables

The size of the set $\{(\bar{b},\bar{b}') \mid B(\bar{b},\bar{b}')\}$ can be exponential in the number of predicates. But in practice, the basic block usually contains just a very small subset of all program variables. The most Boolean variables thus stay unchanged in the abstract version of the basic block. The truth value of the predicate is guaranteed not to change if the predicate uses only variables that are not assigned to. The remaining predicates, which use variables that are assigned to and thus their truth value changes (formally predicates π_i such that $\pi_i(v) \neq \pi_i(v')$) are called *output predicates*. The predicates that influence the truth value of *output predicates* are called *input predicates* and can be obtained by finding out variables used in the assignments to variables mentioned in *output predicates*.

As an optimization, the set $\{(\bar{b},\bar{b}') \mid B(\bar{b},\bar{b}')\}$ can be restricted only to *input* and *output* predicates. It means, that $\bar{b}$ will contain only *input predicates* and $\bar{b}'$ will contain only *output predicates*.

Example 2.9 (input and output predicates):

Let $\pi_1 = i < 5$ and $\pi_2 = j > 0$ are the predicates and let the basic block contain only the statement `i=j;`. Then the predicate π_1 is the only *output predicate* and the predicate π_2 is the only *input predicate*.

3 Confrontation of Bandera, C2BP and SATABS abstraction algorithms

Confrontation of Bandera, C2BP and SATABS abstraction algorithms will be described on theoretical level, because not all of these algorithms are open sources and it is not possible to get their implementation for testing purposes. However, experimental results of these algorithms can be found in relevant papers, e.g. [17] for Bandera, [22] for C2BP and [6] for SATABS.

3.1 Utilization of algorithms

Bandera abstraction algorithm is semi-automatic, it requires the cooperation of the user. The user has to select abstractions for the most important variables. In comparison with Bandera, C2BP and SATABS are fully automatic. However, the approach of Bandera allows supporting all kinds of programs including multi-threaded programs with concurrency. C2BP does not support multi-threaded programs and nor does the basic SATABS algorithm (but there is an extension to SATABS algorithm supporting multi-threaded programs with concurrency, this algorithm is described in [10]).

Bandera was developed for verification of Java programs, therefore the abstractions bound to concrete variables are in form of classes with defined domain and abstract operators. The domain of the abstraction is defined by user, but the abstract versions of operators are automatically generated by Bandera. This approach improves the precision of created abstractions. The cooperation with user can lead to more flexible model; the user is able to select an abstraction optimal for concrete variable. But on the other hand, the user without experience can select unsuitable abstractions which can lead to imprecise model with too many states.

The algorithms C2BP and SATABS work with programs written in the C language. The user can influence the abstraction process only via selection of predicates to be checked. The rest of abstraction process is automatic, therefore the preciseness of the resulting model depends on the predicates which were chosen for abstraction.

3.2 Preciseness of algorithms

Abstractions computed by all three algorithms are over-approximations of concrete programs. It means that all the paths in the concrete programs are also the paths in the abstract programs. But there may exist paths in the abstract programs that are infeasible in the concrete ones.

In Bandera, the program to be abstracted is sliced first. Slicing is the process when all variables that do not influence any of checked properties are removed from the program. It is done according to program dependence graph (more about PDG is in chapter **2.1.5**). Slicing is a very useful process because it is worthy to abstract only variables that can influence at least one of the checked properties. The remaining variables are useless for verification and it would be ineffective to keep them in the program to enlarge the state space.

Since all variables influencing some of the checked properties are included in the resulting program, no information necessary for verification is lost.

Non-determinism (and inferential spurious counterexamples) can appear due to selected abstractions. In some situations, abstractions do not provide enough pieces of information. As an example, consider an abstraction with domain `{pos, neg, zero}` and operation of addition over this domain. Adding "`{pos} + {neg}`" has no concrete result; all the values `{pos, neg, zero}` can serve as possible results. This situation is solved by non-deterministic choice from the set of possible results.

C2BP algorithm creates the Boolean program using only Boolean variables corresponding to predicates arising from properties to be verified. No other variables are included. Dependences between variables are not taken into account.

Non-determinism and spurious counterexamples are caused by lack of information.

As an example, consider a code:

```
typedef struct cell {
      int val;
      struct cell* next;
} *list;

list curr, prev, nextCurr;
```

and the statement

```
curr = nextCurr;
```

to be abstracted. The set of predicates is

```
{curr==NULL, prev==NULL, curr->val > v, prev->val > v}
```

The assignment `curr = nextCurr;` influences the values of Boolean variables `{curr==NULL}` and `{curr->val>v}`, but there is no predicate (and thus no Boolean variable) referring the variable `nextCurr`, so the values of Boolean variables `{curr==NULL}` and `{curr->val>v}` in the Boolean program cannot be determined after the assignment `curr = nextCurr;`. The information about values of both of these Boolean variables is lost. The assignment `curr = nextCurr;` in the concrete program will be replaced by following assignments in the Boolean program:

```
{curr==NULL} = unknown();
{curr->val>v} = unknown();
```

Apparently, the lack of information causes non-determinism and can lead to spurious counterexamples.

SATABS algorithm creates the transition relation using only Boolean variables corresponding to predicates arising from properties to be verified. But the difference in comparison to C2BP is that the transition relation is created with respect to all concrete variables that may influence the Boolean variables.

Non-determinism and spurious counterexamples are caused by dividing the program into smaller blocks called clusters. Transition relations are computed for these individual blocks; the relations among variables are taken into account only within the scope of one block. Therefore, if one variable is related to other one from a different

block, their interference among the blocks cannot be projected into the resulting transition relations of individual blocks. Transition relations have to be over-approximated and that is the opportunity for non-determinism and spurious counterexamples.

In case the program would not be divided into the blocks, instead, it would be taken as a whole as one block, the resulting abstraction (transition relation) would not be an over-approximation but a precise abstraction. The problem of such process is that it would be very ineffective and time intensive. Therefore, the optimization was introduced to compute the transition relations over the blocks (clusters) selected as suitably small parts of the program. Without this optimization, the SATABS algorithm would be unusable for larger programs.

3.3 State space of resulting models

The result of abstraction in Bandera is a program with abstract operations and variables of abstract types. The number of states in the model created using this abstracted program depends on abstractions selected for concrete variables and predicates to be verified. The smaller number of domain values is present in the abstraction interpretation the smaller number of states has the model. The selection of predicates to be verified can influence the number of states in the model by variables referred in these predicates. If the predicates refer to variables that influence all variables in concrete program (no matter via which dependence), then no variables can be sliced and the resulting model contains all variables. And the state space of such model is obviously larger than that of a model with less number of variables.

The result of abstraction in C2BP is a Boolean program, where the only variables are the Boolean variables corresponding to predicates arising from properties to be verified. The number of states of the resulting model thus depends on properties to be verified (rather on the variables referred in corresponding predicates).

If the predicates refer to variables interacting (via mutual assignments) with variables that are not referred in the predicates, the values of Boolean variables corresponding to the predicates will be undecidable (because of loss of information) and enumerated by non-deterministic choice. Such enumeration can enlarge the state space. On the other hand, if the variables referred in the predicates interact (via mutual assignments) with each other, the values of corresponding Boolean variables when assigned by other Boolean variables will have defined exactly one value and thus the state space will be smaller.

The results of abstraction in SATABS are transition relations. The only variables contained in these transition relations are the Boolean variables corresponding to predicates arising from properties to be verified. The number of states of the resulting model depends on properties to be verified. The other factor influencing the number of states is the number of basic blocks and the division of the concrete program to basic blocks itself.

The goal of SATABS is to replace basic blocks in concrete program by transition relations describing what happens to Boolean variables after executing these basic blocks. Each state is represented by values of all Boolean variables in the program (values before the execution of the basic block and after it). The set of states

$\{(\bar{b},\bar{b}') \mid B(\bar{b},\bar{b}')\}$ (see chapter **2.3.2**) can have exponential size in reference to number of predicates. But in praxis, only minimal number of all variables in the program is contained in one basic block and thus only minimal number of Boolean variables values is changed.

3.4 Theorem prover calls

Theorem prover calls belong to main limitations of abstraction processes. The calls are time consuming and the goal of all algorithms is to eliminate the calls as much as possible. Moreover, if the theorem prover is incomplete, checked properties can be undecidable.

Bandera and C2BP use external theorem provers, SATABS keeps clear of theorem provers by using a SAT solver integrated into the process of abstraction. The integration of SAT solver allows more precise abstraction than could be achieved using a theorem prover.

Usage of a theorem prover in Bandera is necessary for computing abstract versions of operators. But this computation is not a part of the abstraction process itself, it is done within the process of abstraction interpretations defining. This process is done independently of the abstraction process, therefore the usage of a theorem prover has no impact on the speed of the abstraction process.

C2BP makes use of a theorem prover in much larger extend than Bandera does. The theorem prover is used directly for creation of abstract Boolean program. In the worst case, there is $O(2^{|E|})$ of theorem prover calls (where E is the set of predicates). There are some optimizations of theorem prover calls number, some of them prefer abstraction correctness and the other prefer the speed of abstraction process. Without optimizations, the algorithm C2BP would be unusable.

First, the theorem prover Simplify supporting only linear arithmetic over real numbers was used, other operators had to be approximated by means of uninterpreted functions. Simplify was then replaced by the theorem prover Zapato that is better than Simplify, but bit-vector arithmetic is still not supported there. Currently, there is a prototype of SLAM using another model checker with SAT solver which can work also with bit-vector constructs.

SATABS does not use the theorem prover, instead, an integrated SAT solver is used for transition relation computation. The SAT solver reduces the number of spurious counterexamples, the resulting abstraction is more precise than the one computed using a theorem prover. Moreover, the tight integration of SAT solver into the processes of abstraction, simulation and abstraction refinement is helpful for precise encoding of the language semantics including pointer arithmetic and bit-vector overflow.

3.5 Supported constructs, features and limitations

Abstraction in Bandera depends on verified property and program dependences. Type inference is used for establishing correct abstractions for all data based on the abstractions chosen by the user. Bandera tools are Java specific, but the abstraction process is not, it is applicable to any program described in terms of JVM byte codes. In order to hide differences among various specification languages, Bandera developed its

own tool-independent language of temporal specification patterns for rendering temporal properties.

Bandera tools enable also an "unsafe" abstraction, which provides another compaction of finite-state models that are useful for bug finding.

The advantage of Bandera compared to C2BP and SATABS is that Bandera supports multi-threaded programs without any restrictions. C2BP cannot work with multi-threaded programs at all, SATABS was already extended to allow abstracting such programs (but originally, SATABS did not support these programs).

Bandera was successfully tested for small and medium Java programs (100 – 1500 code lines). Now the research is being done about the behavior of Bandera over large programs (10 000 – 100 000 code lines).

Speciality of C2BP is that it doesn't insert procedures into the code when abstracting procedure calls. Instead, it computes signatures of the procedures and works with them. Such approach can reduce the code size within the abstraction process and speed up the verification of this code.

C2BP can avoid producing spurious counterexamples where conflicting predicates (e.g. the predicates "`x=2`" and "`x=3`") would hold at once. It is done via an `enforce` construct. Such situation has to be handled because it is not possible to prelude an execution of the Boolean program, where two uninterpreted Boolean variables b_1 and b_2 representing these conflicting predicates are true at the same time. More about the `enforce` construct is in [22].

The disadvantage of C2BP is that the weakest precondition computation is local with respect to abstracted assignment, thus it does not take the control-flow into account. Consequently, the abstraction can be not very precise and this can be source of spurious counterexamples.

Variables are by C2BP modeled using unbounded integers, thus an arithmetic overflow is not considered and checked.

SATABS supports most of ANSI-C constructs including arrays (with possibly unbounded size) and unions. Also, bit-vector operators are supported. Bit-vector semantics is modeled accurately, thus it is possible to detect faults related to bit-level operators and arithmetic overflow.

One block in concrete program is processed as a whole, not as separate variables like in other tools. The relations and interactions among separate variables are therefore also taken into account.

A limitation of SATABS is that a recursion and dynamic memory allocation are not allowed.

3.6 Concluding evaluation

It is not possible to say which algorithm is the best and which is the worst. Each of them serves for different types of programs. Bandera was designed for object oriented

programs (especially java), whereas C2BP and SATABS algorithms serve for abstraction of C programs.

Bandera is only semi-automatic method, but it can work with object oriented and multi-threaded programs. Object oriented approach is totally different from the approach of programming languages like the C language (languages that are not object oriented). Thus it is hard to compare Bandera to C2BP and SATABS algorithms. But it is possible to say that the advantage of Bandera is the ability to work with object oriented programs, because object oriented approach becomes widely used on the field of software development and there are not too many abstraction tools supporting this approach.

To compare the C2BP algorithm with the SATABS algorithm is simpler. Both of these algorithms support programs written in the same programming language. The approach of these algorithms to abstraction is a little bit different. C2BP tries to create a Boolean program from the original one using predicates and a theorem prover whereas SATABS tries to create transition relations for blocks of the original program using predicates and SAT solver.

Theorem prover calls slow down the abstraction process and incompleteness of the theorem prover can even make the abstraction imprecise. The variables are modeled using unbounded integer values; the possible arithmetic overflow is not taken into account. This can lead to false positive answers of the theorem prover. Most of the operators are modeled by means of uninterpreted Boolean functions (only a limited range of operators is supported) what limits the set of programs that can be verified.

SAT solver is integrated right into the abstraction process. The potentially exponential number of theorem prover calls is replaced by an enumeration on a single SAT instance. The tight integration of SAT solver helps to precisely encode language semantics including pointer arithmetic and bit-vector overflow.

Using a SAT solver instead of theorem prover seems to be a good choice. SATABS algorithm uses the SAT solver trying to improve the method of predicate abstraction by eliminating the limitations caused by theorem prover. The number of spurious counter examples caused by SATABS abstraction process is less than for C2BP algorithm.

The other advantage of SATABS algorithm is that its abstraction process can be transformed to support multi-threaded programs the easier way than the abstraction process of C2BP. There is already an extension to SATABS algorithm supporting multi-threaded programs [10]. To the best of our knowledge, there has been no such extension for C2BP algorithm yet. Such extension would be quite complicated and it is not clear if such abstraction could be performed automatically.

Based on the comparison above, it seems that SATABS algorithm is a little bit better than C2BP and it is actually true. The SATABS algorithm was designed later than C2BP trying to improve existing methods for automatic predicate abstraction. The main goal of SATABS project was to replace a theorem prover by an integrated SAT solver which promised fastening and improving of an abstraction process (by removing theorem prover calls, reducing spurious counterexamples and introducing support for bit-vector arithmetic). An aspiration of the SATABS algorithm to improve a method for automatic predicate abstraction was successful.

Benefit of the C2BP algorithm is that it was the first algorithm for automatic abstraction of programs written in the C language and it presented the way how to automatically construct an abstraction. It is a part of the SLAM toolkit designed for verifying that Windows device drivers obey API conventions. This toolkit is successfully being used in praxis and it has already detected some bugs in device drivers. It demonstrates that such tool can be profitably used in praxis.

4 Concept of predicate abstraction for object oriented programs

Chapter **2** describes two algorithms using predicate abstraction - C2BP and SATABS. These algorithms give good results and are fully automatic. However, both of these algorithms can work only with programs that are not object oriented. Today, object oriented programs are widely used for software construction and thus we decided to try to fit the predicate abstraction to object oriented programs.

Abstraction algorithm used in Bandera works with object oriented programs, but it is not fully automatic and the abstraction is not predicate abstraction to all intents and purposes. The resulting abstract program contains variables with modified data types that were defined by user. In contrast to Bandera, we would like to create an abstract program automatically, without any cooperation needed from the user.

An input of our method is an object oriented program *P* and a set of predicates *E*. The program *P* consists of classes and contains variables of basic data types, methods etc. We will not deal with any specific constructs as Exceptions in Java.

First, we remove from the program *P* all data that are independent of the predicates from the set *E* (through data, control, interference, divergence, synchronization or ready dependence). The resulting program is then abstracted using a predicate abstraction. All variables referred in predicates are removed from the program and replaced by Boolean variables representing individual predicates. The result of our abstraction method is an object oriented program containing predicate variables and variables that were not abstracted but cannot be removed because of their dependence on the predicates. If this abstraction is too coarse (i.e. it causes spurious counterexamples), it is refined, predicates that caused spurious counterexamples are added to improve the abstraction and the abstraction process is rerun.

Our abstraction method supports multi-threaded programs, recursion and pointers. We do not explicitly handle these features, but we do not corrupt the structure of the original program and so we do not affect these features by modifying the original program to abstract one.

We speculated whether to create the Boolean program where the only variables are the variables arising from the predicates (i.e. all other variables would be removed from the program). The reasoning why we did not remove the not-abstracted variables from the abstract program is that if we would do so, some information valuable for model checking could be lost. Creating Boolean program as a result of abstraction for object oriented programs is beyond the scope of this work. One has to solve whether all variables can be abstracted and removed or whether some variables have to stay in the program because their removing could corrupt the structure of given program. It is necessary to explore mainly the variables representing individual objects.

4.1 Preconditions

Our concept of abstraction will be specified on a general level to describe main principles of abstraction for object oriented program in complex. This approach does

not allow to abstract concrete programming language constructs specific for individual programming languages.

Our method can work with only some forms of predicates. Basically, we allow two types of predicates:

- Predicates containing only variables from one class (e.g. "`x > 0`", "`y = x + 3`"). The user has to specify the class to which these variables belong.
- Predicates containing variables from two different classes (currently, we cannot handle predicates referring variables from more than two classes; more thorough research has to be done first).

We do not support abstracting of variables used as parameters in procedure calls. The variables referred in the predicates should be of basic data types, we do not support generic types etc.

From now on, an expression "`A.x`" will denote a variable `x` from a class `A`, and an expression "`ObjA.x`" will denote a variable `x` of an object `ObjA` that is an instance of a class `A`. We introduce this marking to avoid problems with distinction between classes and instances in the following text.

4.2 Slicing

Slicing is a process when all parts of program that are independent of the checked properties are removed from the program. This process can significantly reduce the state space of resulting abstract program. Bandera uses Program dependence graph (see chapter **2.1.5**) to detect dependences in program and slices the program according to this graph.

We decided to use slicing to get rid of all unnecessary parts of program before applying predicate abstraction. Parts of code removed by slicing cannot influence the truth values of predicates and the variables from these parts of code would only enlarge the resulting state space. Slicing can be very effective method how to reduce the state space, especially for concurrent object oriented programs. In [16], the effectiveness of slicing for java programs is examined. The costs of slicing are inconsiderable as against the whole verification process.

We will not describe slicing in detail here, it does not form the main part of our abstraction method. More about slicing is in [13] and [15], optimizations of slicing (reduction of PDG edges) is in [24]. We do not aim at designing a new method of slicing, there are several existing tools for slicing that can be used well for our purposes [11].

4.3 Predicate abstraction

Our goal is to design the abstraction of the concrete program the way that would support multi-threaded programs and pointers. In object oriented programs, it is necessary to support pointers and references. We decided to fit the predicate abstraction method of C2BP algorithm (see chapter **2.2**) to object oriented programs. The main difference in comparison with C2BP algorithm is that we will not remove all concrete variables to

create a Boolean program. Some variables will stay not-abstracted what will allow us to support multi-threaded programs and recursion.

4.3.1 Predicates

We can distinguish between two types of predicates:

- Predicates that refer only to variables from one class (lets call them *class predicates*).
- Predicates that refer to variables from more than one class (lets call them *inter-class predicates*).

To compute abstraction regarding the *class predicates* is quite straightforward, it is very similar to computing abstraction by C2BP algorithm.

Computing abstraction using *inter-class predicates* is more difficult. Problems are caused by instances of classes containing variables referred in the predicate. Consider e.g. the predicate "`A.x > B.y`" where `A` and `B` are two different classes. And consider 3 different instances of class `A` and two different instances of class `B`. The truth value of predicate "`A.x > B.y`" can differ for each tupple of instances (`A`, `B`). Thus it is necessary to keep one such predicate for each tupple of instances.

In [19] and [26], this situation is solved by creating a special class for each *inter-class predicate* (each such class is inherited from one common class Abstract). This class contains a variable of two dimensional array type, where each field represents the truth value of the predicate for one given tupple of instances (`A`, `B`). In addition to it, the class also contains methods for getting the field index for given class instance and methods for getting the reference to class instance for given field index. A call to a special method of *inter-class predicate* class is added to the constructor of each class that contains some variables referred in this *inter-class predicate*. This special method adds each new class instance to the array of predicate truth values.

The approach described above currently allows the *inter-class predicates* containing variables from only two different classes, but the research is being done to allow the variables from more than two different classes in one *inter-class predicate*.

Our abstraction method works similarly as in [19]. The variables representing the predicates are added as new variables of Boolean type into relevant classes (in case of *class predicates*) or a new class is created for them (in case of *inter-class predicates*). However, in [19], the abstraction process needs some cooperation with the user; the user has to specify (in addition to predicates) also all variables that are to be removed from the program. In our method, we automatically remove all variables referred in any of the predicates. Our method differs also in the process of assignments and conditions abstraction.

4.3.2 Adding predicates to program

The user is allowed to create predicates either for classes or for class instances. Our approach to both of these types of predicates is about the same. In case of the class predicate (e.g. `A.x > A.y`), the variables referred in the predicate are removed from given class and the predicate is incorporated to the program (in a form of new Boolean

variable added to the class or in a form of new class). In case of class instance predicate (e.g. `ObjA.x > ObjA.y`), the class corresponding to the object `ObjA` has to be found first and the rest of the process is the same as for the class predicate – the variables referred in the predicate are removed from the class and the predicate is incorporated to the program.

We have to discriminate between the *class predicates* and the *inter-class predicates*. The process of adding the predicates into the program differs for these two types of predicates in a way how to incorporate the predicates into the program.

Class predicates:

Consider a predicate "`x > 0`" where `x` is a variable from a class `A`. First, a new variable (lets denote it `XPos`) representing this predicate will be added to class `A` and the variable `x` will be removed from this class. All occurrences of the variable `x` in the program (not only in the class `A`) must be then replaced by the variable `XPos`. E.g. the variable `ObjA.x` (where `ObjA` is an instance of class `A`) has to be replaced by the variable `ObjA.XPos`.

Inter-class predicates:

Consider a predicate "`A.x > B.y`" where `A` and `B` are two different classes. A new class has to be created for this predicate. This class will contain a variable `pred` of two dimensional array type for storing truth values of instances of the predicate "`A.x > B.y`" for individual class instance tupples. Actually, each instance of the predicate "`A.x > B.y`" is a new predicate. This new predicate is represented only by its truth value stored in the corresponding field of the array `pred`. The class will contain the following methods:

- `void setA(A ObjA)` – initializes abstraction variables and sets index for a new instance of class `A`
- `int getA(A ObjA)` – returns the index for class instance `ObjA` in the array with truth values of the predicate
- `A getA(int index)` – returns the class instance `ObjA` corresponding to the given index
- `void setB(B ObjB)` - initializes abstraction variables and sets index for a new instance of class `B`
- `int getB(B ObjB)` - returns the index for class instance `ObjB` in the array with truth values of the predicate
- `B getB(int index)` - returns the class instance `ObjB` corresponding to the given index

4.3.3 Abstraction of assignment statements

Consider an assignment `ObjA.x:=exp;` somewhere in the program P and a set of predicates $E = \{\varphi_1 \ldots \varphi_n\}$. First, we compute the weakest liberal precondition (WP) (see chapter **2.2.3**) for each of the predicates to find out which predicates can be involved by

the assignment `ObjA.x:=exp;`. If the predicate is not involved (i.e., $WP(\texttt{A.x} := \texttt{exp},\ \varphi_i) = \varphi_i$), we will not assign the value to the variable related to this predicate. For each of the predicates that are involved (i.e., $WP(\texttt{A.x} := \texttt{exp},\ \varphi_i) \neq \varphi_i$), we have to add an assignment to the variables related to these predicates. This assignment will be of following form:

$\texttt{VarP} := \texttt{evaluate}\ (\Im_V(WP(\texttt{A.x} := \texttt{exp},\ \varphi_i)),\ \Im_V(WP(\texttt{A.x} := \texttt{exp},\ \neg\varphi_i)))$;

where $\Im_V(\varphi)$ denotes straightening of the predicate φ (see *Formalization of predicate straightening* in chapter **2.2.3**). The function `evaluate()` is defined as follows:

```
bool evaluate(bool pos, bool neg) {
      if (pos) { return true; }
      if (neg) { return false; }
      return random();
}
```

The assignment to variables representing *class predicates* is created straightforwardly, because there is only one such variable for each *class predicate*. For a variable `XPos` representing a predicate "`x > 0`" we get an assignment `XPos := evaluate(…);`.

A little bit more work has to be done when creating an assignment for *inter-class predicates*. Each *inter-class predicate* has as many "copies" (in fact, each copy represents a new predicate) as is the number of tupples (`A`, `B`), where `A` and `B` denote the instances of classes `A` and `B`. Each "copy" of the *inter-class predicate* can have different evaluation and thus we have to compute the weakest liberal precondition (and depending on it also the predicate straightening) for each of these "copies".

Consider the *inter-class predicate* "`A.x > B.y`" and instances `ObjA` and `ObjB` of classes `A` and `B`. A new "copy" of predicate was created for these instances. This "copy" represents the predicate of form "`ObjA.x > ObjB.y`". Actually, the existence of the predicate "`ObjA.x > ObjB.y`" is denoted only by its truth value stored in the field of the array `pred` in class representing the predicate "`A.x > B.y`" (see chapter **4.3.2**).

Consider an assignment `ObjA.x = exp;`. All predicates for all tupples (`ObjA`, `*`) have to be evaluated; we have to create one assignment for each of these predicates. All the assignments have to be processed atomically, because they represent one assignment `ObjA.x = exp;`. The assignment `ObjA.x = exp;` will be replaced by following (pseudo) code:

```
BeginAtomic();
      for(int i = 0; i < number of instances of class B; ++i){
            pred[getA(ObjA)][i] = evaluate(…);
      }
EndAtomic();
```

The parameters for the function `evaluate(…)` will be the same as defined above (straightening of *WP* for an expression `ObjA.x = exp;` and a predicate φ and its negation).

4.3.4 Abstraction of conditions

Ideally, the conditions correspond to some of the predicates and all we have to do is to replace the concrete conditions by the variables representing these predicates. Such situations are quite common, but there can be still some conditions that have to be abstracted in a similar way as assignment statements (using the theorem prover).

Consider a condition C (without loss of generality, we can assume this condition to be of form "$(p_1 \wedge \ldots \wedge p_i) \vee \ldots \vee (p_n \wedge \ldots \wedge p_m)$", where each p_k denotes a predicate – we do not mean the predicate from the set E but a general predicate). If the condition does not refer any variable that was removed from the program, the condition will not be abstracted. In opposite case, the condition will be replaced by the expression "`evaluate` $(\Im_V(C),\ \Im_V(\neg C))$`;`".

4.3.5 Example

Consider a following Java program (Bakery mutual exclusion algorithm):

```
class Process1 extends Thread{
public int y1 = 0;
private Process2 p2;

void SetThread(Process2 p)
{ p2 = p; }

public void run(){
   while (true) {
      y1 = p2.y2 + 1;
      while(p2.y2!=0&&y1>=p2.y2);
      y1 = 0;
}}}
```

```
class Process2 extends Thread{
public int y2 = 0;
private Process1 p1;

void SetThread(Process1 p)
{ p1 = p; }

public void run(){
   while (true) {
      y2 = p1.y1 + 1;
      while(p1.y1!=0&&y2>=p1.y1);
      y2 = 0;
}}}
```

```
class Bakery {
   public static void main(String args[]){
      Process1 proc1 = new Process1();
      Process2 proc2 = new Process2();
      proc1.SetThread(proc2);
      proc2.SetThread(proc1);
      proc1.start();
      proc2.start();
}}
```

A set of predicates that should be verified is as follows:

```
Process1.y1 = 0;
Process1.y1 > 0;
Process2.y2 = 0;
Process2.y2 > 0;
proc1.y1 < proc2.y2;
```

where `Process1` and `Process2` denote classes, `proc1` denotes an instance of the class `Process1` and `proc2` denotes an instance of the class `Process2`.

The first four predicates represent *class predicates*, the last one represents an *inter-class predicate*. Thus we get four Boolean variables for the first four predicates and one new class for the last predicate. Let the corresponding variables (and class) be denoted as `y1IS0`, `y1MORE0`, `y2IS0`, `y2MORE0` and `class y1LESSy2`.

The program (as defined above) together with predicates does not need to be sliced, because no parts of the program would be replaced. The program contains only variables depending on the predicates. Thus we will directly approach an abstraction itself.

The class `y1LESSy2` representing the predicate `proc1.y1 < proc2.y2;` will be defined in following way:

```
     class y1LESSy2 {
(1)    static final int MAX = 3;
(2)    static public boolean[][] pred = new Boolean[MAX][MAX];

(3)    static public int numProcess1 = 0;
(4)    static public Process1[] objProcess1 = new Process1[MAX];
(5)    static public void setProcess1(Process1 obj){
(6)      objProcess1[numProcess1++] = obj; }
(7)    static public int getProcess1(Process1 obj) {
(8)      for(int i = 0; i < numProcess1; ++i)
(9)        if(obj == objProcess1[i]) return i;
(10)     return MAX + 1; }
(11)   static public Process1 getProcess1(int i){
(12)     return objProcess1[i]; }

(13)   static public int numProcess2 = 0;
(14)   static public Process2[] objProcess2 = new Process2[MAX];
(15)   static public void setProcess2(Process2 obj){
(16)     objProcess2[numProcess2++] = obj; }
(17)   static public int getProcess2(Process2 obj) {
(18)     for(int i = 0; i < numProcess2; ++i)
(19)       if(obj == objProcess2[i]) return i;
(20)     return MAX + 1; }
(21)   static public Process2 getProcess2(int i){
(22)     return objProcess2[i]; }
     }
```

An array for storing truth values of the predicate `proc1.y1 < proc2.y2;` for individual class instances of classes `Process1` and `Process2` is represented by a variable `pred[][]`at line `(2)`. The variable `numProcess1` at line `(3)` denotes the number of instances of the class `Process1` created in the program. References to these instances are stored in the individual fields of an array variable `objProcess1[]` (see line `(4)`). Whenever a new instance of the class `Process1` is created, a constructor `setProcess1()` (defined at line `(5)`) has to be called. This constructor increases the value of the variable `numProcess1` and adds a reference to the new instance into the array `objProcess1[]`. Methods for getting an index of a given instance of class `Process1` (`int getProcess1(Process1 obj)`) and for getting a reference to an instance of the class `Process1` stored in the array field with index `i` (`Process1 getProcess1(int i)`) are defined at lines `(7)` and `(11)`. The lines `(13)-(22)` define the same variables and methods as the lines `(3)-(12)` but for the class `Process2`.

The resulting abstract program created using the given predicates will look as follows:

```
     class Process1 extends Thread{
(1)    public boolean y1IS0 = evaluate(true, false);
(2)    public boolean y1MORE0 = evaluate(false, true);
(3)    private Process2 p2;
(4)    void SetThread(Process2 p){ p2 = p; }

(5)    public void run(){
(6)      while(true){
(7)        Verify.BeginAtomic();
(8)        for(int i = 0; i < y1LESSy2.numProcess2; ++i){
(9)          if(i == y1LESSy2.getProcess2(p2))
(10)           y1LESSy2.pred[y1LESSy2.getProcess1(this)][i] =
                Verify.evaluate(false, true);
(11)         else y1LESSy2.pred[y1LESSy2.getProcess1(this)][i] =
                    Verify.evaluate(false,false);
(12)       }
(13)       y1IS0 = Verify.evaluate(false, p2.y2IS0 || p2.y2MORE0);
(14)       y1MORE0 = Verify.evaluate(p2.y2IS0 || p2.y2MORE0, false);
(15)       Verify.EndAtomic();

(16)       while (Verify.evaluate(!p2.y2IS0 &&
               !y1LESSy2.pred[y1LESSy2.getProcess1(this)]
                             [y1LESSy2.getProcess2(p2)],
             p2.y2IS0 ||
             y1LESSy2.pred[y1LESSy2.getProcess1(this)]
                          [y1LESSy2.getProcess2(p2)])
           ) {};

(17)       Verify.BeginAtomic();
(18)       for(int i = 0; i < y1LESSy2.numProcess2; ++i){
(19)         y1LESSy2.pred[y1LESSy2.getProcess1(this)][i] =
              Verify.evaluate(y1LESSy2.getProcess2(i).y2MORE0,
                              y1LESSy2.getProcess2(i).y2IS0 ||
                              !y1LESSy2.getProcess2(i).y2MORE0);
(20)       }
(21)       y1IS0 = Verify.evaluate(true, false);
(22)       y1MORE0 = Verify.evaluate(false, true);
(23)       Verify.EndAtomic();
(24)     }}

(25)   Process1() {y1LESSy2.setProcess1(this)}
     }
```

At lines `(1)` and `(2)`, new variables representing predicates `Process1.y1 = 0` and `Process1.y1 > 0` are introduced. These variables replaced the concrete variable `y1`. Lines `(7)-(15)` show assignments to predicate variables that stand for the concrete assignment `y1 = p2.y2 + 1;`. These assignments are enclosed in an atomic section that has to be computed atomically, because it stands for one assignment. Lines `(8)-(12)` show an assignment to the variable `pred` from the class `y1LESSy2` representing an *inter-class predicate* `proc1.y1 < proc2.y2`. Lines `(13)` and `(14)` show assignments to predicate variables representing predicates `Process1.y1 = 0` and `Process1.y1 > 0`. At line `(16)`, an abstraction of a condition `(y2!=0 && y1>=p2.y2)` is shown. Lines `(17)-(23)` show assignments to predicate variables that stands for the concrete assignment `y1 = 0;` similarly as the lines `(7)-(15)`. Line `(25)` displays a

constructor of the class `Process1` that adds reference to each new instance of this class to the class representing an *inter-class predicate* `proc1.y1 < proc2.y2`.

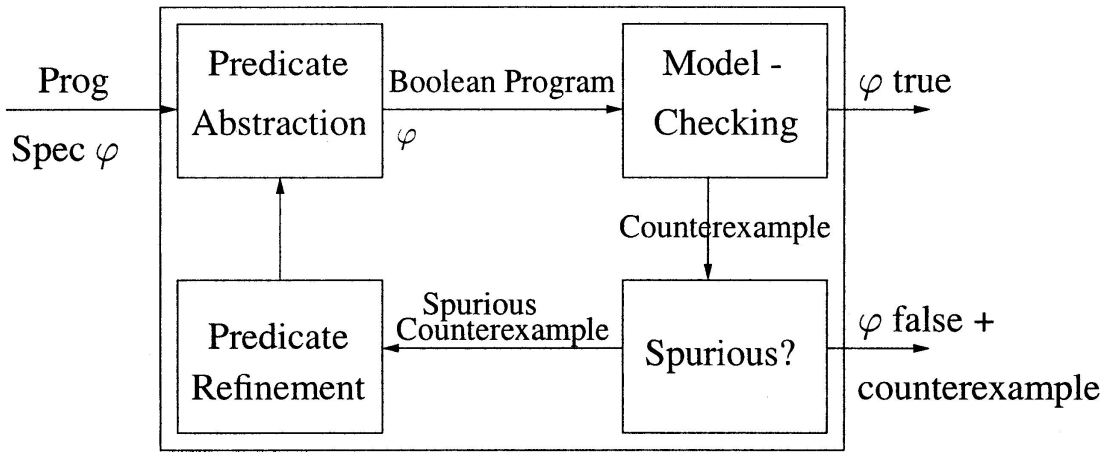
The class `Process2` would be abstracted similarly as the class `Process1` (these classes are about the same), thus we will not explicitly describe it here. The class `Bakery` will stay unchanged; there are no variables that could be abstracted in this class.

4.4 Abstraction refinement loop

When the program is abstracted, it is verified by a model checker. In our case, the tool that can model check the program written in the relevant object oriented programming language has to be used. If a counterexample is found, it has to be examined whether it is a real one or spurious one. In case the counterexample is spurious, it is clear that the abstraction was too coarse and it has to be refined. The process of verifying the program and refining of the abstraction repeats until no spurious counterexample is found. This process is called abstraction refinement loop.

We decided to add an automated abstraction refinement loop into our abstraction method. CounterExample Guided Abstraction Refinement paradigm (CEGAR) [8] is a process for automatic refinement of the abstraction using the spurious counterexample. Simply explained, the counterexample is simulated in the concrete program and when some predicate that holds in the concrete program and does not hold in the abstract program (due to non-determinism) is found, this predicate is added to the set of predicates and the abstraction process is rerun considering this predicate. The abstraction is thus refined by added predicate and the spurious counterexample caused by this predicate will not appear any more.

The CEGAR loop is described clearly in the picture 2.



Picture 2: *The CEGAR loop.*

The abstraction refinement process reduces the amount of non-deterministic choices in the abstract program. Adding a new predicate into the abstraction appends some new piece of information to the abstraction. This piece of information allows the deterministic choice where non-determinism had to be used before.

4.5 Evaluation of our concept of abstraction

Our concept of abstraction works over object oriented programs. Object oriented programs are increasingly used for software development and thus it is necessary to improve methods for model checking of such software. But there is only quite small amount of tools for (automatic) abstraction of object oriented programs. That is the reason why we tried to design a new abstraction method suitable for object oriented programs. Our method can compute the abstraction of these programs automatically.

The size of resulting state space is in our method influenced by predicates, mainly by *inter-class predicates*. The number of these predicates depends on the number of related class instances, we get $|A| \times |B|$ predicates, where $|A|$ is the number of class `A` instances and $|B|$ is the number of class `B` instances. The state space is reduced by slicing variables that do not depend on verified properties and removing variables that were referred in predicates. Removed variables are replaced by predicates though, but the state space of these predicates (of Boolean type) is much smaller than the state space of original variables (of various basic types).

Our concept supports multi-threaded programs, recursion and pointers. If the concrete program was multi-threaded, the abstract program stays multi-threaded. No important information about threads is removed through the abstraction process.

The resulting abstract program is still of the same programming language as the original one. Thus some model checking tool that can work with given programming language is necessary for verifying the abstract program. There are several such tools for different programming languages (e.g. [12] for Java programs), so it would not be problem to fit one of them to our abstraction method.

5 Related work and future research

Our concept of abstraction works with object oriented programs as Bandera [1] does, but in contrary to Bandera, it is designed as a fully automated method. The user can influence the abstraction only by choosing predicates. Bandera allows user to assign variables by abstractions of various types, we induct the predicate variables of Boolean type to reduce the state space.

It is hard to compare our method to C2BP [22] and SATABS [18] algorithms, because these algorithms work over C programs whereas our concept is designed for object oriented programs. Object oriented programs require a little bit different approach. The main problem is necessity to address a distinction between class and instance variables. There can be multiple instances of the classes we want to abstract, which means that multiple predicates must be introduced to perform the abstraction. Abstracted statements must be than generated so that they properly manipulate all of the predicates at once.

We tried to use similar approach to abstraction as in C2BP, but the resulting program is not strictly Boolean program, it contains also not-abstracted variables that are of various basic types. This modification was introduced because it is not clear which variables can be removed from the program and which have to stay there. The variables independent on predicates are removed from the program by slicing before the abstraction process itself and thus only the variables that depend on predicates stay in the program. It is not trivial to decide which of these depending variables could be removed so that no important piece of information would be lost. It would need some more research to find out how to put together the approach of C2BP ("the only variables are the Boolean variables representing predicates") and the object orientedness.

We also considered fitting the SATABS algorithm to object oriented programs, but it would be very difficult and it would need a lot of research first. There would be similar problem as with C2BP – to combine the idea of object orientedness with computation of transition relations.

5.1 Future research

Our concept used the method for working with *inter-class predicates* (described in [19]) that is special for object oriented programs. This method allows only predicates referring variables from at most two different classes. But there is some research to allow variables from more (ideally as much as possible) classes. Such improvement would enlarge the set of predicates that can be used for automatic abstraction of object oriented programs. Much more properties could be then automatically verified in programs written in object oriented languages.

An abstraction methods used by C2BP and SATABS algorithms can significantly reduce the resulting state space. It might be very useful to examine how to fit such (or similar) methods to object oriented approach. First, it is necessary to determine which variables can be replaced by predicates and how.

We tried to use the abstraction method of C2BP in our concept, but we applied only a limited part of it. The result of our abstraction is not a Boolean program as for C2BP, but the program where only some variables are replaced by Boolean variables (related

to predicates). The rest of variables stayed without any abstraction. We decided to use slicing to reduce the number of not abstracted variables. In case the predicate abstraction would be used as strictly as in C2BP, slicing would be unavailing; all not abstracted variables would be removed throughout the process of predicate abstraction.

There has been no implementation of our algorithm yet, because implementing this algorithm would be quite difficult and time-consuming, what is beyond the scope of this work. However, it would be very beneficial to implement this method.

6 Conclusion

In the first part of the work, we described three existing algorithms for (semi-)automatic abstraction of programs, the algorithm of Bandera toolset for abstracting java programs and the C2BP and SATABS algorithms for abstracting programs written in the C language. We compared these methods using the criteria that can demonstrate the advantages, disadvantages and suitability of usage of these methods for different types of programs.

The second part of the work focuses on finding a new method of automatic abstraction for object oriented programs. As far as we know, there is no tool for automatic abstraction of object oriented programs. There are only tools requiring some cooperation with the user [17], [19]. Such cooperation can give very good results, if the user is able to analyze the program and choose an optimal abstraction. However, imagine a programmer who does not understand program verification too much and he needs to verify his program. He would not be able to do it himself without a tool that can verify programs automatically.

We tried to design an algorithm for automatic abstraction of object oriented programs. Our method is based on pieces of knowledge obtained in the first part of the work. The method tries to be general for all object oriented programs (i.e. independent on the concrete programming language). However, there are some restrictions because it is not possible to generalize the abstraction for all various languages.

7 References

[1] Bandera, http://bandera.projects.cis.ksu.edu

[2] Blast model checker for C programs: http://mtc.epfl.ch/software-tools/blast/

[3] Bogor software model checking framework: http://bogor.projects.cis.ksu.edu/

[4] BOOP model checking tool: http://boop.sourceforge.net/

[5] CBMC model checker for ANSI-C programs: http://www.cs.cmu.edu/~modelcheck/cbmc/

[6] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. *Predicate abstraction of ANSI-C programs using SAT.* In Proceedings Of the Model Checking for Dependable Software-Intensive Systems Workshop, San Francisco, USA, 2003.

[7] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. *SATABS: SAT-based predicate abstraction for ANSI-C.* In: Tools and Algorithms for Construction and Analysis of Systems, 2005.

[8] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu and Helmut Veith. *Counterexample-guided abstraction refinement.* In Computer Aided Verification, 2000.

[9] Hassen Saïdi. *Model checking guided abstraction and analysis.* In Proceedings of the 7th International Static Analysis Symposium (SAS'00), Lecture Notes in Computer Science, 2000.

[10] Himanshu Jain, Daniel Kroening, Edmund Clarke. *Verification of SpecC using Predicate Abstraction.* Proceedings of MEMOCODE 2004.

[11] Indus (tool for slicing of Java programs): http://indus.projects.cis.ksu.edu/

[12] Java PathFinder model checking tool: http://javapathfinder.sourceforge.net/

[13] John Hatcliff, James Corbet, Matthew B. Dwyer, Stefan Sokolowski and Hongjun Zheng. *A formal study of slicing for multi-threaded programs with JVM concurrency primitives.* Technical Report 99-6, Kansas State University, Department of Computing and Information Sciences, March 1999.

[14] Ken McMillan. *Symbolic model checking.* Kluwer Academic Publishers, 1993.

[15] Matthew B. Dwyer, James C. Corbet, John Hatcliff, Stefan Sokolowski and Hongjun Zheng. *Slicing multi-threaded java programs: a case study.* Technical report 99-7, Kansas State University, Department of Computing and Information Sciences, March 1999.

[16] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Ranganath, Robby, and Todd Wallentine. *Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs.* In: TACAS 2006, LNCS 3920, pp. 73-89, 2006.

[17] Matthew B. Dwyer, John Hatcliff, Robby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Robby, Hongjun Zheng, Willem Visser. *Tool-supported program abstraction for finite-state verification.* In: ICSE 01: Software Engineering (to appear), 2001.

[18] SATABS: http://www.verify.ethz.ch/satabs/

[19] SeungJoon Park, Willem Visser and Phil Oh. *Predicate abstraction for object-oriented programs.* Published at http://ic.arc.nasa.gov/ic/publications/ in 10/26/2000.

[20] SLAM project (including C2BP algorithm): http://research.microsoft.com/slam

[21] SPIN model checker: http://spinroot.com/spin/whatispin.html

[22] Thomas Ball, Rupak Majumdar, Todd Millstein and Sriram K. Rajamani. *Automatic predicate abstraction of C programs.* In: PLDI 01: Programming Language Design and Implementation, ACM (2001) 203-213.

[23] Thomas Ball, Sriram K. Rajamani. *Bebop: A Symbolic Model Checker for Boolean Programs*. In Proceedings of the 7th SPIN Workshop, Stanford University, California, 2000

[24] Venkatesh Prasad Ranganath and John Hatcliff. *Honing the detection of interference and ready dependence for slicing concurrent java programs.* Technical Report, SAnToS -TR2003-6, October 2003.

[25] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park and Flavio Lerda. *Model checking programs.* Automated Software Engineering Journal, Volume 10, Number 2, April 2003.

[26] Willem Visser, SeungJoon Park and John Penix. *Using predicate abstraction to reduce object-oriented programs for model checking.* In FMSP 00: Formal Methods in Software Practice, pages 3-12. ACM, 2000.

8 Appendix

A Example of C2BP abstraction

Consider the following program for list partitioning written in the C language:

```
typedef struct cell {
   int val;
   struct cell* next;
} *list;

list partition(list *l, int v) {
   list curr, prev, newl, nextCurr;

   curr = *l;
   prev = NULL;
   newl = NULL;
   while (curr != NULL) {
      nextCurr = curr->next;
      if (curr->val > v) {
         if (prev != NULL) {
            prev->next = nextCurr;
         }
         if (curr == *l) {
            *l = nextCurr;
         }
         curr->next = newl;
         newl = curr;
      } else {
         prev = curr;
      }
      curr = nextCurr;
   }
   return newl;
}
```

A set of predicates that are to be verified for this program is:

```
curr==NULL;
prev==NULL;
curr->val > v;
prev->val > v;
```

An abstract program generated by C2BP algorithm will look like follows (inline comments show the code of the original program that was replaced by an abstract one):

```
void partition() {
  bool {curr==NULL}, {prev==NULL};
  bool {curr->val>v}, {prev->val>v};
  {curr==NULL} = unknown();          //curr = *l;
  {curr->val>v} = unknown();
  {prev==NULL} = true;               //prev = NULL;
  {prev->val>v} = unknown();
  skip;                              //newl = NULL;
  while(*) {                         //while(curr!=NULL)
    assume(!{curr==NULL});
    skip;                            //nextCurr = curr->next
    if (*) {                         //if(curr->val > v) {
      assume({curr->val>v});
      if (*) {                       //if(prev!=NULL)
        assume(!{prev==NULL});
        skip;                 //prev->next= nextCurr;
      }                       //}
      if (*) {                //if(curr == *l) {
        skip;                 //*l = nextCurr;
      }                       //}
      skip;                   //curr->next = newl;
      skip;                   //newl = curr;
    } else {                  //} else {
      assume(!{curr->val>v});
      {prev==NULL} = {curr==NULL}; //prev = curr;
      {prev->val>v} = {curr->val>v};
    }                         //}
    {curr==NULL} = unknown();        //curr = nextCurr;
    {curr->val>v} = unknown();
  }
  assume({curr==NULL});
}
```

B Example comparing Bandera abstraction to our abstraction

In this example, we will create two variants of abstract program for one concrete program. The first variant will be created using the Bandera abstraction algorithm; the second variant will be created using our concept of abstraction. We do not aim to demonstrate which method is a better one. Our goal is to illustrate differences between these two approaches to abstraction of object oriented programs.
The program to be abstracted is the following Java program:

```
class Event{
   int count = 0;
   public synchronized void wait_for_event(){
      try{wait();}
      catch(InterruptedException e){};
   }
   public synchronized void signal_event(){
      count = count + 1;
      notifyAll();
} }

class FirstTask extends Thread{
   Event event1,event2;
   int count = 0;
   public FirstTask(Event e1, Event e2){
      this.event1 = e1; this.event2 = e2;
   }
   public void run(){
      count = event1.count;
      while(true){
         if (count == event1.count)
            event1.wait_for_event();
         count = event1.count;
         event2.signal_event();
} } }

class SecondTask extends Thread{
   Event event1,event2;
   int count = 0;
   public SecondTask(Event e1, Event e2){
      this.event1 = e1; this.event2 = e2;
   }
   public void run(){
      count = event2.count;
      while(true){
         event1.signal_event();
         if (count == event2.count)
            event2.wait_for_event();
         count = event2.count;
} } }

class START{
   public static void main(String[] args){
      Event event1 = new Event();
      Event event2 = new Event();
      FirstTask task1 = new FirstTask(event1,event2);
      SecondTask task2 = new SecondTask(event1,event2);
      task1.start(); task2.start();
} }
```

We are trying to verify if the program satisfies following properties:

```
FirstTask.count == event1.count
SecondTask.count == event2.count
```

where `FirstTask` and `SecondTask` denote classes, `event1` and `event2` denote instances of the class `Event`.

In case of Bandera, various abstract programs can be created. An abstraction is influenced by a choice of the user. He can decide between more possible abstractions for individual variables. One of such abstract programs can be the program where the variables `count` of an integer type will be replaced in both classes `Event` and `FirstTask` by the variables `count` of `Signs` type (for more information about this type, see chapter **2.1.3**).

The abstract program looks like following:

```
class Event {
   int count = Signs.ZERO;
   public synchronized void wait_for_event(){
      try {wait();}
      catch(InterruptedException e){};
   }
   public synchronized void signal_event(){
      count = Signs.add(count,Signs.POS);
      notifyAll();
}}

class FirstTask extends Thread {
   Event event1, event2;
   int count = Signs.ZERO;
   public FirstTask(Event e1, Event e2){
      this.event1 = e1; this.event2 = e2;
   }
   public void run() {
      count = event1.count;
      while (true){
         if(Signs.eq(count,event1.count))
            event1.wait_for_event();
         count = event1.count;
         event2.signal_event();
}}}
```

The class `SecondTask` will be abstracted similarly as the class `FirstTask`. The class `START` will stay unchanged, because it does not contain any variables suitable for abstraction.

In case of our abstraction method, an abstract program creation is automatic and thus only one variant of an abstract program can be computed.

A new class `FcntEQEcnt` representing an inter-class predicate `FirstTask.count == event1.count` has to be added:

```
class FcntEQEcnt {
  static final int MAX = 3;
  static public boolean[][] pred = new boolean[MAX][MAX];

  static public int numFirstTask = 0;
  static public FirstTask[] objFirstTask = new FirstTask[MAX];
  static public void setFirstTask(FirstTask obj){
    objFirstTask[numFirstTask++] = obj;
  }
  static public int getFirstTask(FirstTask obj){
    for(int i = 0; i < numFirstTask; ++i)
      if(obj == objFirstTask[i]) return i;
    return MAX + 1;
  }
  static public FirstTask getFirstTask(int i){
    return objFirstTask[i];
  }

  static public int numEvent = 0;
  static public Event[] objEvent = new Event[MAX];
  static public void setEvent(Event obj){
    objEvent[numEvent++] = obj;
  }
  static public int getEvent(Event obj){
    for(int i = 0; i < numEvent; ++i)
      if(obj == objEvent[i]) return i;
    return MAX + 1;
  }
  static public Event getEvent(int i){
    return objEvent[i];
  }
```

Similarly, we have to add a new class `ScntEQEcnt` representing an inter-class predicate `SecondTask.count == event2.count`. This class will look about the same as the class `FcntEQEcnt` and thus we will not specify it here.

The remaining classes of the abstract program will look like follows:

```
class Event{
   public synchronized void wait_for_event(){
      try { wait();}
      catch(InterruptedException e){}
   }
   public synchronized void signal_event(){
      Verify.BeginAtomic();
      for(int i = 0; i < FcntEQEcnt.numFirstTask; ++i){
         FcntEQEcnt.pred[i][FcntEQEcnt.getEvent(this)] =
                Verify.evaluate(false, FcntEQEcnt.pred[i]);
      }
      for(int i = 0; i < ScntEQEcnt.numSecondTask; ++i){
         ScntEQEcnt.pred[i][ScntEQEcnt.getEvent(this)] =
                Verify.evaluate(false, ScntEQEcnt.pred[i]);
      }
      Verify.EndAtomic();
      notifyAll();
   }
   Event() {
      FcntEQEcnt.setEvent(this);
      ScntEQEcnt.setEvent(this);
} }

Class FirstTask extends Thread {
   Event event1, event2;
   Verify.BeginAtomic();
   FcntEQEcnt.pred[this][FcntEQEcnt.getEvent(event1)] =
                Verify.evaluate(false, false);
   Verify.endAtomic();

   public void run() {
      Verify.BeginAtomic();
      FcntEQEcnt.pred[this][FcntEQEcnt.getEvent(event1)] =
                Verify.evaluate(true, false);
      Verify.endAtomic();
      while (true){
         if(Verify.evaluate(FcntEQEcnt.pred[this]
                     [FcntEQEcnt.getEvent(event1)]))
            event1.wait_for_event();
         Verify.BeginAtomic();
         FcntEQEcnt.pred[this][FcntEQEcnt.getEvent(event1)] =
                   Verify.evaluate(true, false);
         Verify.endAtomic();
         event2.signal_event();
}}}
```

The class `SecondTask` will be abstracted similarly as the class `FirstTask`. The class `START` will stay unchanged, because it does not contain any variables suitable for abstraction. Thus we will not explicitly describe the abstraction of these classes here.